Broadview
www.broadview.com.cn



跟老齐学Python从入门到精通

齐伟 编著



人生苦短，我用Python
零基础起步，手把手进阶，辅以实际案例，
Python这样学才简单！





 中国工信出版集团

 電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT...

LANZAROTE Caliente.com

Broadview®
www.broadview.com.cn





跟老齐学Python从入门到精通

齐伟 编著



人生苦短，我用Python
零基础起步，手把手进阶，辅以实际案例，
Python这样学才简单！






中国工信出版集团


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

版权信息

书名：跟老齐学Python：从入门到精通
作者：齐伟
出版社：电子工业出版社
ISBN：978-7-121-28034-4
定价：69.00

版权所有·侵权必究

前言

这是一本学习材料，是为编程“零基础”的朋友学习Python提供的类似教材的学习材料，所以，内容会有庞杂琐碎之感，但这对于“零基础”的读者来讲是不可缺少的。所以，不要把这本书当作“开发手册”来用。

本书虽然是以“零基础”起步，但是并不打算仅仅涉及一些浅显的入门知识，当然基础知识是必不可少的，还想为“零基础”的朋友多提供一些知识，一些所谓高级的内容，既满足了好奇心，也可以顺势深入研究。当然，真正的深入还需要读者自己努力。

“敬畏上帝是智慧的开端”。在本书的编写过程中，一直惶恐于能否所言无误，但水平有限，错误难免，敬请读者指出，并特别建议，对有异议的地方，请使用Google网站搜索更多的资料进行比较阅读，也可以跟我联系，共同探讨。为了便于进行技术交流，我创建了一个QQ群（群号：26913719），专供本书读者研讨技术问题。

完成本书是一个比较漫长的过程，在这个过程中，得到了很多朋友的帮助，在这里对他们表示感谢，并将他们的名号列在下面：

李航、令狐虫、github641、dongm2ez、wdyggh、codexc、winecat、solarhell、ArtinHuang、吴优。

在本书编辑过程中，电子工业出版社的编辑高洪霞、黄爱萍为本书的面世做出了极大的努力，对她们的工作表示诚挚感谢。

最后，要感谢我的妻子，在本书的写作过程中，她给了我很多鼓励，还协助我检查文本内容。

希望这本书能够为有意学习Python的读者提供帮助。

齐伟

2016年1月

第1季　基础

从这里开始，请读者——你已经确信自己是要学习Python的准程序员了——跟我一起，领略一番Python的基础知识，这是学好Python的起步，同时，其内容也和其他的高级编程语言有相通之处。所以，学习Python是一种“性价比”非常高的事情。

在本季中，要向读者介绍Python的基本对象类型、语法规则和函数的相关知识。学习完这些内容，就能够用Python做很多事情了，且在其中还会不断强化一种掌握Python的方法。

第0章　预备

从现在开始，本书将带领你——零基础的学习者——进入到Python世界。进入这个世界，你不仅能够体会到Python的魅力，感受到编程的快乐，还顺便可以成为一个程序员，我相信你一定能成为一个伟大的程序员，当然这并不是本书的目的，更不是本书的功劳。当你成为一个技术大牛的时候，最应该感谢的是你的父母，如果你顺便也感谢一下本书，比如多购买一些本书分发给你的弟兄们，那是我的福份，感激不尽。

预备，Let’s go！

0.1 关于Python的故事

学习一种编程语言是一件很有意思的事情，从现在开始，我就和你一起来学习一种叫作Python的编程语言。

在编程界，存在着很多某种语言的忠实跟随者，因为忠实，就会如同卫道士一样有了维护那种语言荣誉的义务，所以总见到有人争论哪种语言好、哪种语言不好。当然，好与坏的标准是不一样的，有些人以学了之后能不能挣大钱为标准，有些人以是否容易学为标准，或许还有人以能不能将来和妹子一同工作为标准（也或许没有），甚至有些人就没有什么标准，只是凭感觉或者道听途说而人云亦云罢了。

读者在本书中将看到一个颇为迷恋于Python的人，因为全书看不到一句有关Python的坏话（如果有，则肯定是笔误，是应该删除的部分）。

不管是语言还是其他什么，挑缺点是比较容易的事情，但找优点都是困难的，所以，《圣经》中那句话——为什么你看见弟兄的眼中有刺，却不想自己眼中有梁木呢？——是值得我们牢记的。

在本书开始就废话连篇，显见本书不会有什么“干货”，倒是“水货”颇多，并不是因为“水是生命的源泉”，而是因为作者水平有限，如果不掺“水”，唯恐说不清道不明，还敬请读者谅解。嫌“水”多的，就此可以合上本书去看网上的各种电影吧。也不用在网上喷我，因为那样只能增加更多的“口水”（还是水）。

下面说点儿正经的。

0.1.1 Python的昨天、今天和明天

这个题目有点大了，似乎回顾过去、考察现在、张望未来都是那些掌握方向的大人物做的。那就让我们每个人都成为大人物吧。因为如果不回顾一下历史，似乎无法满足好奇心；如果不考察一下现在，也不放心（担心学了之后没有什么用途）；如果不张望一下未来，怎么能吸引（也算是一种忽悠吧）你呢？

1.Python的历史

历史向来是成功者的传记，现在流传的关于Python的历史也是如此。

Python的创始人为吉多·范罗苏姆（Guido van Rossum），关于他开发Python的过程，很多资料里面都要记录下面的故事：

1989年的圣诞节期间，吉多·范罗苏姆为了在阿姆斯特丹打发时间，决心开发一个新的脚本解释程序，作为ABC语言的一种继承。之所以选中Python作为程序的名字，是因为他是一个蒙提·派森的飞行马戏团的爱好者。ABC是由吉多参加设计的一种教学语言，在吉多本人看来，ABC这种语言非常优美和强大，是专门为非专业程序员设计的。但是ABC语言并没有成功，究其原因，吉多认为是非开放造成的。吉多决心在Python中避免这一错误，并取得了非常好的效果，完美结合了C和其他一些语言。

这个故事是从维基百科里面直接复制过来的，很多讲Python历史的资料里面，也都转载了这一段文字。但是，在我来看，吉多是为了“打发时间”而决定开发Python，源自他的这样一段自述：

Over six years ago，in December 1989，I was looking for a"hobby"programming project that would keep me occupied during the week around Christmas.My office（a government-run research lab in Amsterdam）would be closed，but I had a home computer，and not much else on my hands.I decided to write an interpreter for the new scripting language I had been thinking about lately:a descendant of ABC that would appeal to Unix/C hackers.I chose Python as a working title for the project，being in a slightly irreverent mood（and a big fan of Monty Python's Flying Circus）.（原文地址：https://www.python.org/doc/essays/foreword/）

首先，必须承认，这个哥们儿是一个非常牛的人，此处献上恭敬的崇拜。

其次，刚刚开始学习Python的朋友，可千万别认为Python是一个可以随随便便鼓捣的东西，人家也是站在巨人的肩膀上的。

第三，牛人在成功之后，往往把奋斗的过程描绘得比较简单。或者是出于谦虚，或者是为了让人听起来他更牛。反正，我们看最后结果的时候，很难感受过程中的酸甜苦辣。

不管怎样，吉多·范罗苏姆在那时刻创立了Python，而且，更牛的在于他具有现代化的思维——开放，并通过Python社区，吸引来自世界各地的开发者，参与Python的建设。在这里，请读者一定要联想到Linux和它的创始人林纳斯·托瓦兹。两者都秉承“开放”思想，得到了来自世界各地开发者和应用者的欢呼和尊敬。

请读者向所有倡导“开放”的牛人们表示敬意，是他们让这个世界变得更美好，他们以行动诠释了热力学第二定律——“熵增原理”。

2.Python的现在

Python现在越来越火了，因为它搭上了“大数据”、“云计算”、“自然语言处理”等这些时髦名词的便车。

网上时常会有一些编程语言排行榜之类的东西，有的初学者常常被排行榜所迷惑，总想要学习排列在第一位的，认为排在第一位的编程语言需求量大。不管排行榜是怎么编制的，Python虽然没有登上状元、榜眼、探花之位，但也不太靠后呀。

另外一个信息，更能激动一下初学者那颗脆弱的小心脏。

Dice.com网上对20000名IT专业人士进行调查的结果显示：Java类程序员平均工资91060美元；Python类程序员平均工资90208美元。

Python程序员比Java程序员的平均工资低，但看看差距，再看看两者的学习难度，学习Python绝对是一个性价比非常高的投资。

这么合算的编程语言不学等待何时？

3.Python的未来

Python的未来要靠读者了，你学好了、用好了，未来它就光明了，它的未来在你手里。如图0-1所示为Python创始人吉多·范罗苏姆。

0.1.2　Python的特点

很多高级语言都宣称自己是简单的、入门容易的，并且具有普适性，但真正能做到这些的，只有Python。有朋友做了一件衬衫，上面写着“生命有限，我用Python”，这说明什么？说明Python有着简单、开发速度快、节省时间和精力的特点。因为它是开放的，有很多可爱的开发者（为开放社区做贡献的开发者是最可爱的人），将常用的功能做好了放在网上，谁都可以拿过来使用。这就是Python，这就是开放。



图0-1　Python创始人：吉多·范罗苏姆

恭敬地抄录来自《维基百科》的描述：

Python是完全面向对象的语言，函数、模块、数字、字符串都是对象，并且完全支持继承、重载、派生、多继承，有益于增强源代码的复用性。Python支持重载运算符，因此也支持泛型设计。相对于Lisp这种传统的函数式编程语言，Python对函数式设计只提供了有限的支持。有两个标准库（functools和itertools）提供了Haskell和Standard ML中久经考验的函数式程序设计工具。

虽然Python可能被粗略地分类为“脚本语言”（Script Language），但实际上一些大规模软件开发项目（例如Zope、Mnet、BitTorrent及Google）也都广泛地使用它。Python的支持者较喜欢称它为一种高级动态编程语言，原因是“脚本语言”泛指仅做简单程序设计任务的语言，如shell script、VBScript等，但其只能处理简单任务的编程语言，并不能与Python相提并论。

Python本身被设计为可扩充的，并非所有的特性和功能都集成到语言核心。Python提供了丰富的API和工具，以便程序员能够轻松地使用C、C＋＋、Cython来编写扩充模块。Python编译器本身也可以被集成到其他需要脚本语言的程序内。因此，很多人还把Python作为一种“胶水语言”（glue language）使用，使用Python将其他语言编写的程序进行集成和封装。在Google内部的很多项目，例如Google Engine使用C＋＋编写性能要求极高的部分，然后用Python或Java/Go调用相应的模块。《Python技术手册》的作者马特利（Alex Martelli）说：“2004年，Python已在Google内部使用，Google招募许多Python高手，但在这之前就已决定使用Python。他们的目的是尽量使用Python，在不得已时改用C＋＋；在操控硬件的场合使用C＋＋，在快速开发时使用Python。”

可能这里面有一些术语还不是很理解，没关系，只要明白：Python是一种很牛的语言，应用简单，功能强大，Google都在使用，这就足够了，足够让你下决心学习了。

0.1.3　Python哲学

Python之所以与众不同，还在于它强调一种哲学理念：优雅、明确、简单。有一段诗歌读起来似乎很玄，但真实反映了Python开发者的开发理念：

```
The Zen of Python
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than “right” now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

网上能够看到这段文字的中文译本，读者可以去搜索阅读。

0.2 从小工到专家

这个标题，我借用了一本书的名字——《程序员修炼之道：从小工到专家》，并在此特别推荐阅读。

“从小工到专家”也是很多刚学习编程的朋友的愿望。如何能实现呢？《程序员修炼之道：从小工到专家》这本书中，给出了非常好的建议，值得借鉴。

有一个学习Python的朋友曾问我：“书已经看了，书上的代码也运行过了，习题也能解答了，但是还不知如何开发一个真正的应用程序，不知从何处下手，怎么办？”

另外，也遇到过一些刚刚毕业的大学生，从简历上看，相关专业的考试分数是不错的（我一般相信那些成绩是真的），但是，一讨论到专业问题，常常不知所云，特别是当让他面对真实的工作对象时，表现出来的比成绩单差太多了。

对于上述情况，我一般会武断地下一个结论：练得少。

要从小工成长为专家，必经之路是要多阅读代码，多调试程序。古言“拳不离手，曲不离口”，多练习是成为专家的唯一途径。

0.2.1 零基础

有一些初学者，特别是非计算机专业的人，担心自己基础差，不能学好Python。诚然，在计算机方面的基础越好，对学习任何一门新的编程语言越有利。但如果是“绝对零基础”也不用担心，本书就是从这个角度切入来满足你的需要的。凡事总得有一个开始，那么就让本书成为你学习编程语言的开始吧。

就我个人来看，Python是比较适合作为学习编程的入门语言的。

美国有不少高校也这么认为，他们纷纷用Python作为编程专业甚至是非编程专业的大学生入门语言，如图0-2所示为美国各高校设立的编程语言专业。

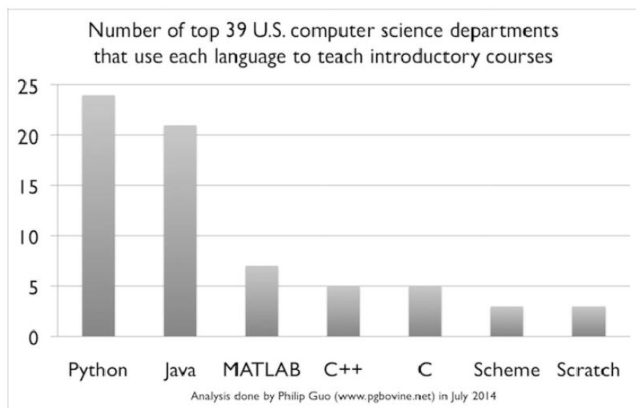


图0-2　美国高校设立的编程语言专业

总而言之，学习Python，你不用担心基础问题。

0.2.2　阅读代码

有句话说得好："读书破万卷，下笔如有神"，这也适用于编程。通过阅读别人的代码，"站在巨人的肩膀上"，让自己眼界开阔，思维充实。

阅读代码的最好地方就是：www.github.com。

如果你还没有账号，请尽快注册，它将是你成为一个优秀程序员的起点。当然，不要忘记来follow我，我的账号是：qiwsir。

阅读代码最好的方法是一边阅读、一边进行必要的注释，这样可以梳理对别人代码的认识。然后可以run一下，看看效果。当然，还可以按照自己的设想进行必要修改，然后再run。经过几轮，就可以将别人的代码消化吸收了。

0.2.3　调试程序

阅读是信息的吸收过程，写作则是信息的加工输出过程。

要自己动手写程序。"一万小时定律"在编程领域也是成立的，除

非你天生就是天才，否则，只有通过“一万小时定律”才能成为天才。

在调试程序的时候，要善于应用网络，看看类似的问题别人是如何解决的，不要仅局限于自己的思维范围。利用网络就少不了使用搜索引擎，在此特别向那些要想成为专家的小工们说：Google能够帮助你成为专家。

我不相信“三天掌握Python”、“三周成为高手”之类的让人听起来热血沸腾、憧憬无限的骗人宣传。如果你通过本书跟我对话，至少说明你我都是普通人，普通人要做好一件事情，除了“机缘巧合”遇到贵人之外，就要靠自己勤学苦练了，没有捷径，凡是宣传捷径的，大多都是骗子。

0.3 安装Python

任何高级语言都需要一个自己的编程环境，这就好比写字一样，需要有纸和笔，在计算机上写东西，也需要有文字处理软件，比如各种名称的Office类软件。笔和纸以及Office软件，就是写东西的硬件或软件，总之，那些文字只能写在相应的硬件或软件上，才能最后成为一篇文章。编程也要有个程序之类的东西，把代码写在上面，才能形成类似文章那样的文件——自己编的程序。

阅读本书的零基础朋友，乃至于非零基础的朋友，不要希望在这里学到很多高深的Python语言技巧。重要的是学会一些方法，比如刚才给大家推荐的“上网Google一下”，就是非常好的学习方法。互联网的伟大之处，不仅仅在于打游戏、看看养眼的照片或者各种视频之类的，我衷心希望本书的读者不仅仅把互联网当作娱乐网，还要当作知识网和创造网。

扯远了，拉回来。在学习过程中，遇到一点点疑问都不要放过，思考、尝试之后，不管有没有结果，都要Google一下，当然，要做到这一点，需要有点技术，这点技术对于立志于成为编程专家的读者来说是必须要掌握的。如果阅读到这里，你还没有理解，那说明的确是我的读者：零基础。

《辟邪剑谱》和《葵花宝典》这两本盖世武功秘籍，对练功者提出了一个较高的要求：欲练神功，挥刀自宫。

学Python，如果要达到比较高的水平，其实不用自宫（如果读者真的要以此明志，该行为结果与本书的作者和出版机构无关，纯属个人行为。若读者尚未成年，请在法定监护人的监护下阅读本书，特此声明）。但是，需要在你的计算机中安装一些东西，并且这个过程有时比较耗费时间。

所需要安装的东西，都在这个页面里面：www.python.org/downloads/。

www.python.org是Python的官方网站，如果你的英语非常好，那么阅读该网站，可以获得非常多的收获。

在下载页面里面，显示出Python目前有两大版本：Python 3.x.x

和Python 2.7.x。可以说，Python 3是未来，它比Python 2.7有进步。但是，现在还有一些东西没有完全兼容Python 3，所以，本书在讲解中以Python 2.7为主，兼顾Python 3.x。一般而言，如果学了Python 2.7，再学习Python 3就很容易，因为两者只是某些地方的变化。请读者不要纠结于是学习Python 2还是学习Python 3这种无聊的问题，如果两个差别达到让你学了而这个无法使用那个，这将是Python的灾难。绝顶聪明的Python创造者和维护者们，绝对不会允许这样的事情出现。

0.3.1 Ubuntu系统

你的计算机是什么操作系统的？如果是Linux的某个发行版，比如Ubuntu，那么就跟我同道了。如果是iOS，也一样，因为都是UNIX麾下的，与在Ubuntu中的安装过程大同小异。只是Widows有点另类了。

但没关系，Python就是跨平台的，它可以在任何系统下进行编程，用它写出来的程序也可以运行在任何操作系统中。

但是，我个人推荐使用Linux/UNIX系列的操作系统。

正常情况下，只要安装了Ubuntu这个操作系统，就已经安装好了Python的编程环境。你只需要打开Shell，然后输入Python，单击“回车”之后就会看到如下内容：

```
$ python
Python 2.7.6 (default, Nov 13 2013, 19:24:16)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more informat
>>>
```

如果没有该编程环境，就需要安装了，一种最简单的方法：

```
$ sudo apt-get install python
```

还有另外一种比较麻烦的方法，但似乎能炫耀一下自己的水平，方法如下。

（1）到官方网站下载源码。比如：

```
wget http://www.python.org/ftp/python/2.7.8/Python-2.7.8.tgz
```

（2）解压源码包

```
tar -zxvf Python-2.7.8.tgz
```

（3）编译

```
cd Python-2.7.8 ./configure --prefix=/usr/local
make&&sudo make install
```

这里指定了安装目录/usr/local。如果不指定，可以使用默认的，直接运行./configure即可。

安装好之后，进入Shell，输入Python，就会看到结果。我们将那个带有“＞＞＞”的界面称之为“Python的交互模式”。

0.3.2 Windows系统

到下载页面里找到Windows安装包下载，比如下载了这个文件：python-2.7.8.msi。然后完成安装。

特别注意，安装完之后，需要检查系统环境变量是否有Python，如果没有，就设置一下。

以上搞定后，在cmd中，输入“python”，得到与前面类似的结果，就说明已经安装好了。

0.3.3 Mac OS X系统

其实根本就不用再写怎么安装Mac OS X系统了，因为用Mac OS X的朋友，肯定是高手中的高高手了，至少我一直很敬佩那些用Mac OS X并坚持没有更换为Windows的人。

所幸的是用苹果电脑的就不用安装了，因为里面一定预装好了，拿过来就可以直接用。

如果按照以上方法顺利安装成功，只能说明幸运。如果没有安装

成功，则是提高自己的绝佳机会，因为只有遇到问题才能解决问题，才能知道更深刻的道理，不要怕，使用Google，它能帮助你解决问题。当然，也可以加入QQ群或者通过微博问我。

0.4　集成开发环境（IDE）

安装好Python之后，就可以进行开发了。按照惯例，第一行代码总是：Hello World。

0.4.1　值得纪念的时刻：Hello world

不管你使用的是什么操作系统，总之肯定能够找到一个地方运行Python，进入到交互模式，像下面一样：

```
Python 2.7.6 (default, Nov 13 2013, 19:24:16)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more informat
>>>
```

在“＞＞＞”后面输入print"Hello，World"，并按回车键，下面是见证奇迹的时刻。

```
>>> print "Hello, World"
Hello, World
```

如果你从来不懂编程，那么从这一刻起，就跨入了程序员行列；如果已经是程序员，那么就温习一下当初的惊喜吧！

“Hello，World”是你用代码向这个世界打招呼了。

每个程序员，都曾经经历过这个伟大时刻，不经历这个伟大时刻的程序员不是伟大的程序员。为了纪念这个伟大时刻，理解其伟大之所在，下面执行分解动作：

说明：在下面的分解动作中，用到了符号“#”，这个符号在Python编程中表示注释。所谓注释，就是在计算机中不执行那句话，只是为了说明某行语句表达什么意思，是给计算机前面的人看的。特别提醒，在编程实践中，注释是必须的，尽管很多人要求代码要具有可读性，但必要的注释也是少不了的。请牢记：程序在大多数情况下是给人看的，只是偶尔让计算机执行一下。

#看到“＞＞＞”符号，表示Python做好了准备，等待你向它发出指令，让它做事情。

```
>>>
```

#print，意思是打印。在这里也是这个意思，是要求Python打印东西。

```
>>> print
```

#"Hello,World"是打印的内容，注意，双引号是英文状态下的。引号不是打印内容，它相当于一个包裹，把打印的内容包起来，统一交给Python。

```
>>> print "Hello, World"
```

#上面命令执行的结果。Python接收到你要求它所做的事情：打印Hello,World，于是它就老老实实地执行这个命令，丝毫不走样。

```
Hello, World
```

在Python中，如果进入了上面的样式，就是进入了“交互模式”。这是非常有用而且简单的模式，便于我们进行各种学习和探索，随着学习的深入，你将更加觉得它魅力四射。

笑一笑：有一个程序员，感觉自己书法太烂了，于是立志继承光荣文化传统，购买了笔墨纸砚。在某天开始练字，将纸铺好，拿起笔蘸足墨水，挥毫在纸上写下了两个大字：Hello World。

虽然进入了程序员序列，但是，如果程序员用这个工具仅仅是打印“Hello，World”，又怎能用“伟大”来形容呢？况且这个工具也太简陋了。你看美工妹妹用的Photoshop，行政妹妹用的Word，出纳妹妹用的Excel，就连坐在老板桌后面的那个家伙也在用PPT播放自己都不相信的新理念呢，难道我们伟大的程序员，就用这么简陋的工具来写旷世代码吗？

当然不是。软件是谁开发的？程序员。程序员肯定会先为自己打造好用的工具，这也叫作“近水楼台先得月”。

IDE就是程序员的工具。

0.4.2　集成开发环境概述

IDE的全称是：Integrated Development Environment，简称IDE，也称为Integration Design Environment或Integration Debugging Environment，翻译成中文叫作“集成开发环境”，它是一种辅助程序员开发用的应用软件。

维基百科这样对IDE定义：

IDE通常包括程序语言编辑器、自动建立工具和除错器。有些IDE包含编译程序和直译器，如微软的Microsoft Visual Studio，有些则不包含，如Eclipse、SharpDevelop等，这些IDE是通过调用第三方编译器来实现代码的编译工作的。有时IDE还会包含版本控制系统和一些可以设计图形用户界面的工具。许多支持面向对象的现代化IDE还包括类别浏览器、对象查看器、对象结构图。虽然目前有一些IDE支持多种程序语言（例如Eclipse、NetBeans、Microsoft Visual Studio），但是一般而言，主要还是针对特定的程序语言而量身打造（例如Visual Basic）。

如图0-3所示是微软提供的名字叫作Microsoft Visual Studio的IDE，用C#进行编程的程序员都用它。

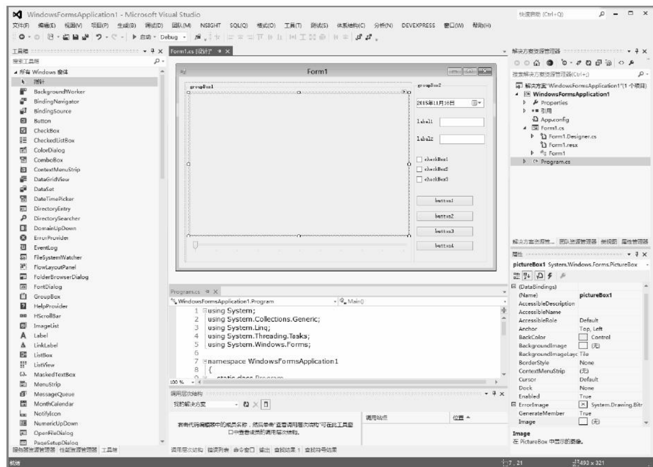


图0-3　名叫Microsoft Visual Studio的IDE

如图0-4所示是在苹果电脑中出现的名叫XCode的IDE。

要想了解更多IDE的信息，推荐阅读维基百科中的词条。

- 英文词条：Integrated development environment
- 中文词条：集成开发环境

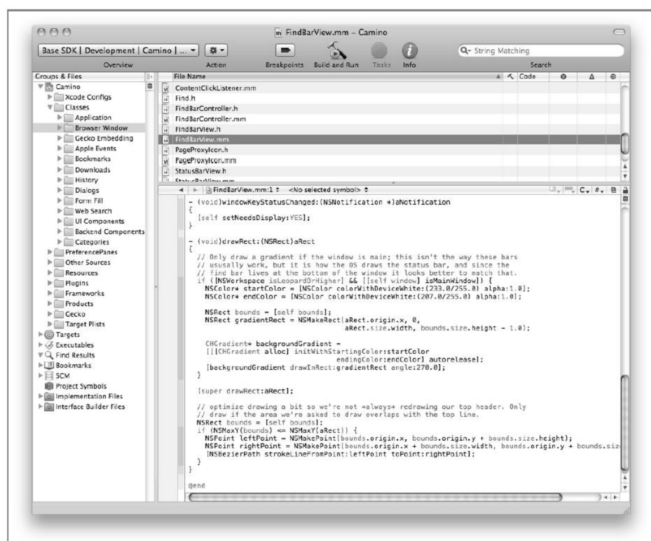


图0-4　名叫XCode的IDE

0.4.3　Python的IDE

用Google搜索一下：Python IDE，会发现能够进行Python编程的IDE还真不少。东西一多就容易无所适从，所以有不少人都问用哪个IDE好。可以看看下面链接里的内容：http://stackoverflow.com/questions/81584/what-ide-to-use-for-python。

顺便推荐一个非常好的与开发相关的网站：stackoverflow.com。在这里可以提问，可以查看答案。如果有问题，一般先在这里查找，大多数情况都能找到非常满意的结果，且有很大启发。

那么作为零基础的学习者，用哪个IDE好呢？既然是零基础，就别瞎折腾了，就用Python自带的IDLE，原因就是：简单，虽然它比较简陋。

在Windows中，通过“开始”菜单→“所有程序”→“Python

2.x”→“IDLE（Python GUI）”来启动IDLE。启动之后，看到如图0-5所示的界面。

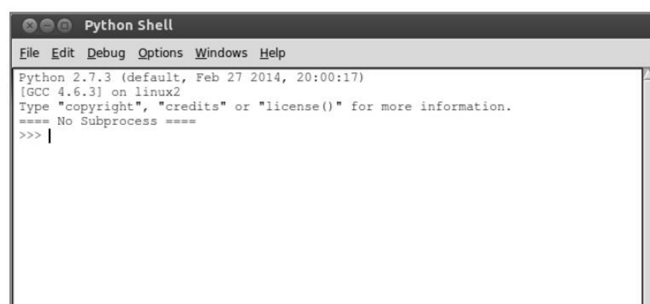


图0-5　IDLE界面

注意：若看到的界面显示版本与这个图不同，则说明安装的版本不同，但大致模样差不多。

其他操作系统的用户，也都能在启动IDLE这个程序之后，出现与上面一样的图。

这里其实是Python的交互模式编程环境。

当然还有一个文本环境，读者可以在File菜单中新建一个文件，即进入文本编辑环境。

除了这个自带的IDE，还有很多其他的IDE，列出来，供喜欢折腾的朋友参考。

- PythonWin:是Python Win32 Extensions（半官方性质的Python for win32增强包）的一部分，也包含在ActivePython的Windows发行版中。如其名字所言，只针对Win32平台。
- MacPython IDE：MacPythonIDE是Python的Mac OS发行版内置的IDE，可以看作是PythonWin的Mac对应版本，由Guido的哥哥Just van Rossum编写。
- Emacs和Vim：Emacs和Vim号称是这个星球上最强大的文本编辑器，对于许多程序员来说是万能IDE的不二选择。
- Eclipse＋PyDev：Eclipse是新一代的优秀泛用型IDE，虽然是基于Java技术开发的，但出色的架构使其具有不逊于Emacs和Vim的可扩展性，现在已经成为了许多程序员最爱的瑞士军刀。

磨刀不误砍柴工，IDE已经有了，伟大程序员就要开始从事伟大的编程工作了。

第1章　基本的对象类型

在Python中，“万物皆对象”，在一开始就这么提出来，如果你没有理解，也不用为此吃不好、睡不好，就当听了一个新名词，因为对于“对象”的理解，可以说要贯穿全书，甚至贯穿你的代码生涯。随着实践的增多，对于“万物皆对象”的领悟会逐渐深化。

什么是对象？现在暂时不给出定义，一定要等到时机成熟的时候给出定义才能理解。

如果要准确描述Python对象，需要从“身份、类型、值”三个维度来描述，这三个维度也构成了一个对象的特征。至于怎样从上述三个维度来描述对象，本章将以Python中基本的对象类型为例进行说明。如果换一种说法，也可以理解为本章将带领你了解Python中的某些常用的和基本类型的对象。

1.1　数字

现在更多人把计算机叫作电脑，英文是Computer。只要提到它，普遍都会想到它能够比较快地做加减乘除，甚至乘方、开方等各种数学运算。

有一篇名为《计算机前世》的文章（参见：http://www.flickering.cn），这样讲到：

先来看看计算机（Computer）这个词是怎么来的。英文学得好的小伙伴看到Computer，第一反应是："compute-er"，应该是个什么样的人吧，对，"做计算的人"。叮咚！恭喜你答对了。最先被命名为Computer的确实是人。也就是说，电子计算机（与早期的机械计算机）被赋予这个名字是因为他们执行的是被分配到的人的工作。"计算机"原来是工作岗位，它被用来定义一个工种，其任务是执行计算，诸如导航表、潮汐图表、天文历书和行星的位置要求等的重复计算，从事这个工作的人就是Computer，而且大多是女神。

原文还附有如图1-1所示的图片。

所以，以后要用第三人称来称呼Computer，请用She（她）。现在你明白为什么程序员中那么多"他"了吧，因为Computer是"她"。



图1-1　从事Computer工作的人

1.1.1 数字

在Python中，对数的规定比较简单，达到了小学数学水平即可理解。

首先，进入到Python的交互模式中，不管是什么操作系统，相信你总能找到那个交互模式。然后模仿下面的操作：

```
>>> 3
3
>>> 3333333333333333333333333333333333333333
3333333333333333333333333333333333333333L
>>> 3.222222
3.222222
```

在交互模式下，如果输入3，就显示了3，这样的数称为整数，这个称呼和小学数学一样。

当输入一个比较大的整数时，Python会自动将这个大整数进行转换，转换的结果是一个“长整数”类型，为了表示它，会在其末尾显示一个L。由于这个操作是Python自动完成的，所以在现在的Python中，没有单独将“长整数”作为一个类型。

3.222222在数学里面称为小数，这里依然可以这么称呼，不过就像很多编程语言一样，习惯称之为“浮点数”。至于这个名称的由来，也是有点说道的，有兴趣可以搜索一下。

上述举例中都无符号（或者说是非负数），如果要表示负数，跟数学中的表示方法一样，前面填上负号即可。

值得注意的是，我们这里说的都是十进制的数。

除了十进制，还有二进制、八进制、十六进制都是在编程中可能用到的，这些知识不作为本书讲解的内容，读者要了解，可以寻找相关书籍，或者去网上搜索。

每个数字在Python中都是一个对象，比如前面输入的3就是一个对象。每个对象，在内存中都有自己的一个地址，这就是它的身份。

```
>>> id(3)
140574872
>>> id(3.222222)
```

```
140612356
>>> id(3.0)
140612356
>>>
```

用内建函数id()可以查看每个对象的内存地址，即身份。

内建函数，英文为built-in Function，读者根据名字也能猜个八九不离十了。不错，就是Python中已经定义好的内部函数。

以上三个不同的数字是三个不同的对象，具有三个不同的内存地址。特别要注意，在数学上，3和3.0是相等的，但是在这里，它们是不同的对象。

用id()得到的内存地址是只读的，不能修改。

了解了“身份”，再来看“类型”，也有一个内建函数供使用，即type()。

```
>>> type(3)
<type 'int'>
>>> type(3.0)
<type 'float'>
>>> type(3.222222)
<type 'float'>
```

＜type‘int’＞说明3是整数类型（Interger）；＜type‘float’＞则告诉我们该对象是浮点型（Floating point real number）。与id()的结果类似，type()得到的结果也是只读的。

至于对象的值，在这里就是对象本身了。

看来对象也不难理解。

请保持自信，继续。

1.1.2 变量

仅仅写出“3、4、5”是远远不够的，在编程中，经常要用到“变量”和“数”（严格来讲是对象）建立起对应关系。例如：

```
>>> x = 5
>>> x
5
>>> x = 6
>>> x
6
```

在这个例子中，x＝5就是在变量x和数5之间建立了对应关系，接着又建立了x与6之间的对应关系。我们可以看到，x先“是”5，后来“是”6。

在Python中，有这样一句话是非常重要的：对象有类型，变量无类型。怎么理解呢？

对象的类型，可以使用type()来查看，前面已经演示了。

当在Python中写入了5、6，Computer就自动在其内存中某个地方建立了这两个对象，就好比建造了两个雕塑，一个形状似5，一个形状似6，这两个对象的类型就是Int.

那个x就好比是一个标签，当x＝5时，就是将x这个标签拴在了5上，通过这个x，就顺延看到了5，于是在交互模式中，“＞＞＞x”输出的结果就是5，给人的感觉似乎是x就是5，而事实是x这个标签贴在5上面。同样的道理，当x＝6时，标签就换位置了，贴到6上面。

所以，这个标签x没有类型之说，它不仅可以贴在整数类型的对象上，还能贴在其他类型的对象上，比如后面会介绍到的str（字符串）类型的对象等。

Python中变量的这个特点（即可以四处乱贴的标签）非常重要，它没有类型。

1.1.3　简单的四则运算

读者可以在交互模式中复习一下小学数学中的四则运算，并且报告给你小学数学老师，他（她）当初煞费苦心的教育成果在这里得到了应用。

```
>>> 2+5
7
>>> 5-2
```

```
3
>>> 10/2
5
>>> 5*2
10
>>> 10/5+1
3
>>> 2*3-4
2
```

上面的运算中，分别涉及四个运算符号：加（+）、减（-）、乘（*）、除（/）

另外，我相信读者已经发现了一个重要的公理：

计算机中的四则运算和小学数学中学习过的四则运算规则是一样的

要不怎么说人是高等动物呢，自己发明的东西，一定要继承自己已经掌握的知识，别跟自己的历史过不去。伟大的科学家们，在当初设计计算机的时候就想到我们现在学习的需要了，一定不能让后世子孙再学新的运算规则，就用小学数学里面的好了。感谢那些科学家先驱者，泽被后世。

下面计算3个算术题，看看结果是什么：

```
4 + 2
4.0 + 2
4.0 + 2.0
```

你可能愤怒了，这么简单的题目，就不要劳驾计算机了，太浪费了。

别着急，还是要运算一下，然后看看结果有没有不一样，要仔细观察哦。

```
>>> 4+2
6
>>> 4.0+2
6.0
>>> 4.0+2.0
6.0
```

不一样的地方是：第一个公式结果是6，这是一个整数；后面两

个是6.0，这是浮点数。

计算机做一些四则运算是不在话下的，但是，有一个问题请务必注意：在数学中，整数是可以无限大的，但是在计算机中，整数不能无限大。

因此，就会有某种情况出现，就是参与运算的数或者运算结果超过了计算机中最大的数了，这种问题称之为“整数溢出问题”。

1.1.4　整数溢出问题

在网上能够找到很多专门讨论“整数溢出”问题的文章。

在某些高级编程语言中，整数溢出是必须正视的，但是，在Python里面就无需忧愁了，原因就是Python为我们解决了这个问题，它支持“无限精度”的整数，所以，不用考虑整数溢出的问题，Int类型与任意精度的Long整数类可以无缝转换，超过Int范围的情况都将转换成Long类型。

体验一下大整数：

```
>>> 123456789870987654321122343445567678890098876*123345566778999O
1522784771935275628700443525875762772775623283620324443390191589373
```

多么幸运呀，Python做了如此精妙的安排，免除了麻烦，所以选择学习Python就是珍惜光阴。

你还可以在交互模式下计算2的1000次幂，计算方法是：

```
>>> 2 ** 1000
```

看看结果是什么？你会感到惊喜的。

上面计算结果的数字最后有一个L，表示这个数是一个长整数，你不用管它，反正是Python为我们搞定了大整数问题。

1.2　除法

用单独一个章节来说明除法，就是因为它常常会带来麻烦，不仅Python会这样，很多高级语言都如此。

1.2.1　整数与整数相除

进入Python交互模式之后，练习下面的运算：

```
>>> 2 / 5
0
>>> 2.0 / 5
0.4
>>> 2 / 5.0
0.4
>>> 2.0 / 5.0
0.4
```

看到了吗？麻烦出来了（这是在Python 2.x中），按照数学运算，以上四个运算结果都应该是0.4。但我们看到第一个结果居然是0。Why？

在Python（严格说是Python 2.x中，Python3会有所变化）里面有一个规定，像2/5这样的除法要取整（就是去掉小数，但不是四舍五入）。2除以5，商是0（整数），余数是2（整数）。如果用这种形式：2/5，那么计算结果就是商那个整数。或者可以理解为：整数除以整数，结果是整数（商）。

比如：

```
>>> 5 / 2
2
>>> 7 / 2
3
>>> 8 / 2
4
```

再次提醒：得到是商（整数），而不是得到含有小数位的结果再通过“四舍五入”得到整数。例如：5/2，得到的商是2，余数是1，最终5/2＝2，并不是对结果进行四舍五入得到3。

1.2.2　浮点数与整数相除

“浮点数与整数相除”用一种貌似严格的语言表述：

假设：x除以y。其中x可能是整数，也可能是浮点数；y可能是整数，也可能是浮点数，但两者之中至少有一个是浮点数。

出结论之前，还是先在交互模式中做实验：

```
>>> 9.0 / 2
4.5
>>> 9 / 2.0
4.5
>>> 9.0 / 2.0
4.5
>>> 8.0 / 2
4.0
>>> 8 / 2.0
4.0
>>> 8.0 / 2.0
4.0
```

就如同做物理、化学实验一样，仔细观察上面的实验结果，能得出什么结论？

不管是被除数还是除数，只要有一个数是浮点数，结果就是浮点数。

然而，下面的实验可能又让你有点糊涂了：

```
>>> 10.0 / 3
3.3333333333333335
```

这个是不是就有点搞怪了？按照数学知识，应该是3.33333...，后面是3的循环了，那么你的计算机就停不下来了，满屏都是3。为了避免这个，Python武断终结了循环，但是，可悲的是没有按照“四舍五入”的原则终止。当然，还会有更奇葩的出现：

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.1 - 0.2
0.0
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> 0.1 + 0.1 + 0.1 - 0.2
0.10000000000000003
```

越来越糊涂了，为什么Computer姑娘在计算这么简单的问题上，如此糊涂了呢？

不是Computer姑娘糊涂，她依然冰雪聪明。

原因在于十进制和二进制的转换上，Computer姑娘用的是二进制进行计算，上面的例子中，我们输入的是十进制，就要把十进制的数转化为二进制，然后再计算。但是，在转化中，浮点数转化为二进制，就出问题了。

例如十进制的0.1，转化为二进制是：0.0001100110011001100110011001100110011001100110011...

也就是说，转化为二进制后，不会精确等于十进制的0.1。同时，计算机存储的位数是有限制的，所以，就出现了上述现象。

这种问题不仅仅在Python中有，所有支持浮点数运算的编程语言都会遇到。

明白了问题原因，怎么解决呢？就Python的浮点数运算而言，大多数计算机每次计算误差不超过2的53次方分之一。对于大多数任务这已经足够了，但是要在心中记住这不是十进制算法，每个浮点数计算可能会带来一个新的舍入错误。

一般情况下，只要简单地将最终显示的结果用“四舍五入”到所期望的十进制位数，就会得到期望的最终结果。

对于需要非常精确的情况，可以使用decimal模块（关于“模块”，后面会介绍，这里暂存），它实现的十进制运算适合高精度要求的应用。另外fractions模块支持另外一种形式的运算，它实现的运算基于有理数（因此像1/3这样的数字可以精确地表示）。最高要求则是使用由SciPy提供的Numerical Python包和其他用于数学和统计学的包。列出这些东西，仅仅是让读者明白，问题已经解决，并且方

式很多。

1.2.3 引用模块解决除法问题

Python之所以受人欢迎，一个很重重要的原因就是“轮子”多，当然这是比喻，就好比你要跑得快，怎么办？光天天练习跑步也是不行的，还要用轮子。找辆自行车，就快了很多，若还嫌不够快，再换电瓶车、汽车、高铁……反正可以供你选择的很多。但是，这些让你跑得快的东西，多数不是你自己造的，是别人造好了你来用。甚至两条腿也是感谢父母恩赐。正是因为轮子多，可以选择的多，就可以有各种不同的速度享受了。

轮子是人类伟大的发明。

Python就是这样，有各种“轮子”供我们选用。只不过那些“轮子”在Python里面的名字不叫自行车、汽车，而叫“模块”，有的还叫作“库”、“类”。

怎么用？可以通过以下两种形式。

形式1：import module-name。import后面跟空格，然后是模块名称，例如：import os。

形式2：from module1 import module11。module1是一个大模块，里面还有子模块module11，只想用module11，就这么写。

找一个解决除法问题的轮子：

```
>>> from __future__ import division
>>> 5 / 2
2.5
>>> 9 / 2
4.5
>>> 9.0 / 2
4.5
>>> 9 / 2.0
4.5
```

引用了模块之后再做除法，那么不管什么情况，都能得到浮点数的结果了。

这就是“轮子”的力量。

1.2.4 余数

前面计算5/2的时候，商是2，余数是1

余数怎么得到？在Python中（其实大多数语言也如此），用%符号来取得两个数相除的余数。

实验下面的操作：

```
>>> 5 % 2
1
>>> 6 % 4
2
>>> 5.0 % 2
1.0
```

利用符号“%”可以得到两个数（可以是整数，也可以是浮点数）相除的余数。

除了利用“%”符号之外，还可以使用内建函数，完成同样的工作。

```
>>> divmod(5,2)   #表示5除以2，返回了商和余数
(2, 1)
>>> divmod(9,2)
(4, 1)
>>> divmod(5.0,2)
(2.0, 1.0)
```

内建函数divmod()返回的是两个值，这两个值在一个圆括号内，圆括号内的数字第一个表示商，第二个表示余数。

1.2.5 四舍五入

“四舍五入”在运算中是经常遇到的，按照我们已经对Python的理解，其应该提供一个简单的方法。的确是，有一个内建函数：round()。

```
>>> round(1.234567, 2)
1.23
>>> round(1.234567, 3)
1.235
>>> round(10.0/3, 4)
3.3333
```

在round()中的第二个数，表示要保留的小数位数，返回值是一个四舍五入之后的数值。

简单吧？越简单的时候，越要小心，当你遇到下面的情况，就会有点儿怀疑：

```
>>> round(1.2345,3)
1.234                #应该是：1.235
>>> round(2.235,2)
2.23                 #应该是：2.24
```

哈哈，我发现了Python的一个Bug，太激动了。

别那么激动，如果真的是Bug，还这么明显，是轮不到我的。为什么？具体解释看这里，下面摘录官方文档中的一段话：

Note:The behavior of round()for floats can be surprising:for example，round（2.675，2）gives 2.67 instead of the expected 2.68.This is not a bug:it’s a result of the fact that most decimal fractions can’t be represented exactly as a float.See Floating Point Arithmetic:Issues and Limitations for more information.

原来真的轮不到我。归根到底还是浮点数中的十进制转化为二进制惹的祸。

除法的问题似乎要到此结束了，其实远远没有，不过，作为初学者，至此即可。

1.3　常用数学函数和运算优先级

在数学之中，除了加减乘除四则运算之外（这是小学数学），还有其他更多的运算，比如乘方、开方、对数运算等，要实现这些运算，需要用到Python中的一个模块：math。

1.3.1　使用math模块

math模块是Python标准库中的，所以不用安装就可以直接使用。使用方法是：

```
>>> import math
```

用import就将math模块引用过来了，下面就可以使用这个模块提供的工具了。比如，要得到圆周率：

```
>>> math.pi
3.141592653589793
```

这个模块都能做哪些事情呢？可以用下面的方法看到：

```
>>> dir(math)
    ['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin'
```

dir（module）是一个非常有用的指令，可以通过它查看任何模块中所包含的工具。从上面的列表中就可以看出，在math模块中，可以计算正弦sin()、余弦cos()、开平方0sqrt()等函数。

这些函数是math中提供的，不需要我们编写，可以拿过来就用。比如计算乘方，可以使用pow函数。但是，怎么用呢？

Python是一个非常周到的“姑娘”，她早就提供了一个命令，让我们来查看每个函数的使用方法。

```
>>> help(math.pow)
```

在交互模式下输入上面的指令，然后回车，看到下面的信息：

```
Help on built-in function pow in module math:
pow(...)
    pow(x, y)
    Return x**y (x to the power of y).
```

这里非常清楚地告诉了我们math中的pow函数的使用方法和相关说明。

由于是第一次用到help()，有必要将结果详细解释一番。

1）第一行意思是说这里是math模块的内建函数pow的帮助信息（built-in称之为内建函数，是说这个函数是Python默认的）。

2）第三行表示这个函数的参数，有两个，也是函数的调用方式。

3）第四行是对函数的说明，返回x**y的结果，并且在后面解释了x**y的含义。

最后，按q键返回到Python交互模式。

当然，也看到了一个额外的信息，就是pow函数和x**y是等效的，都是计算x的y次方。

```
>>> 4 ** 2
16
>>> math.pow(4,2)
16.0
>>> 4 * 2
8
```

特别注意，4**2和4*2是有很大区别的。

用help()函数，可以查看math模块中任何一个函数的使用方法。

下面是几个常用的math模块中函数举例，你可以结合自己调试的进行比照。

```
>>> math.sqrt(9)
3.0
>>> math.floor(3.14)
```

```
3.0
>>> math.floor(3.92)
3.0
>>> math.fabs(-2)      #等价于abs(-2)
2.0
>>> abs(-2)
2
>>> math.fmod(5,3)     #等价于5%3
2.0
>>> 5 % 3
2
```

在上面的内容中，读者除了要了解math模块外，还要体会对后续学习非常有帮助的内建函数dir()和help()。

1.3.2　两个函数

下面两个也是常用的数学函数。

1.求绝对值

```
>>> abs(10)
10
>>> abs(-10)
10
>>> abs(-1.2)
1.2
```

2.四舍五入

```
>>> round(1.234)
1.0
>>> round(1.234, 2)
```

如果不清楚这个函数的用法，可以使用下面的方法看帮助信息。

```
>>> help(round)
Help on built-in function round in module __builtin__:
round(...)
    round(number[, ndigits]) -> floating point number
    Round a number to a given precision in decimal digits (default
```

```
This always returns a floating point number.  Precision may be
```

1.3.3　运算优先级

从小学数学开始，就研究运算优先级的问题，比如在四则运算中“先乘除，后加减”，说明乘法和除法的优先级要高于加法和减法。对于同一级别的，就按照“从左到右”的顺序进行计算。

下面的表格中列出了Python中的各种运算的优先级顺序。不过，就一般情况而言，不需要记忆，完全可以按照数学中的运算规则去理解，因为人类既然已经发明了数学，在计算机中进行的运算就不需要重新编写一套新规范了，只需要符合数学中的运算规则即可。

在此读者只需要看个大概、有个印象即可，可以一边向后阅读，一边回来翻阅，或者在用到的时候来这里查看。如表1-1所示。

表1-1　运算规则

运　算　符	描　　述
lambda	Lambda 表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”
in，not in	成员测试
is，is not	同一性测试
<，<=，>，>=，!=，==	比较
|	按位或
^	按位异或
&	按位与
<<，>>	移位
+，-	加法与减法
*，/，%	乘法、除法与取余
+x，-x	正负号
~x	按位翻转
**	指数

最后要提及的是运算中的绝杀：括号。只要有括号，就先计算括号里面的。这是数学中的共识，无需解释。

1.4 第一个简单的程序

通过对四则运算的学习，已经初步接触了Python中的内容，如果是零基础的学习者，可能会有点迷惑：难道敲几个命令，然后看到结果，就算编程了？这也不是那些能够自动运行的程序啊？

到目前为止，还不能算编程，只能算会用一些指令（或者叫作命令）来做点儿简单的工作。

少安毋躁，下面就开始编写一个真正的、简单的程序。

1.4.1 程序

下面一段内容是关于程序的概念，内容来自维基百科：

计算机程序（Computer Program）是指一组指示计算机或其他具有信息处理能力、装置每一步动作的指令，通常用某种程序设计语言编写，运行于某种目标体系结构上。打个比方，一个程序就像一个用汉语（程序设计语言）写下的红烧肉菜谱（程序），用于指导懂汉语和烹饪手法的人（体系结构）来做这个菜。

通常，计算机程序要经过编译和链接而成为一种人们不易看清而计算机可解读的格式，然后运行。未经编译就可运行的程序，通常称之为脚本程序（script）。

简而言之，程序就是指令的集合。但是，有的程序需要编译，有的不需要。Python编写的程序就不需要编译，因此她也被称之为解释性语言，编程出来的程序叫作脚本程序。

某些程序员认为“编译型语言比解释性语言高价”，这是错误的。不要认为编译的程序好，不编译的就不好；也不要认为编译的程序属于“高端”，不编译的就属于“低端”，这是毫无根据的。

不争论，用得妙就是好。

1.4.2　用IDE编程

在实践中，每个人都有自己喜欢的IDE。所以，也请读者在学习中找到自己喜欢的IDE，

Python有一个默认IDE，当打开了Python Shell之后，通过“File->New window”打开一个文本编辑界面。如图1-2所示。在这个界面中就可以写程序了。

在这个界面看不到用于输入指令的提示符“>>>”，这个界面有点像记事本。说对了，其本质上就是一个记事本，只能输入文本，不能直接在里面贴图片。如图1-3所示。

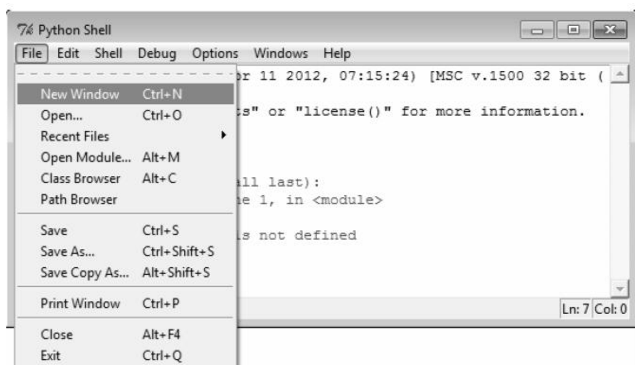


图1-2　文本编辑界面

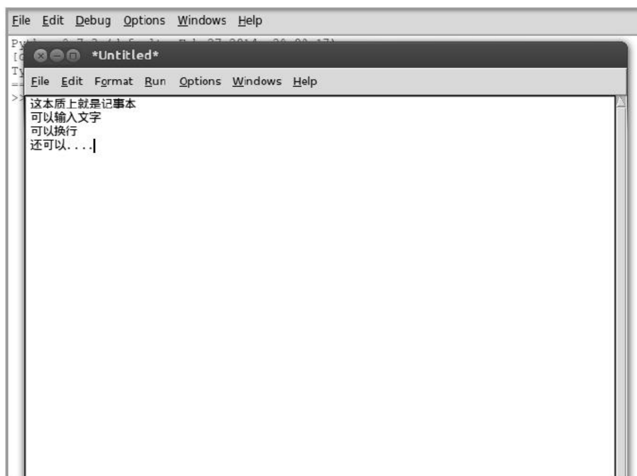


图1-3　输入界面

1.4.3　Hello，World

“Hello，World”是面向世界的标志，所以，任何程序的第一句一定要写这个，因为程序员是面向世界的，绝对不畏缩在某个局域网内。

直接上代码，就这么一行即可。

```
print "Hello,World"
```

如图1-4所示为代码样式。

程序就是指令的集合。现在，这个程序里面就一条指令，一条指令也可以成为集合。

注意观察，单击Run菜单，在下拉列表里面选择“Run Module”，如图1-5所示。



图1-4　代码样式

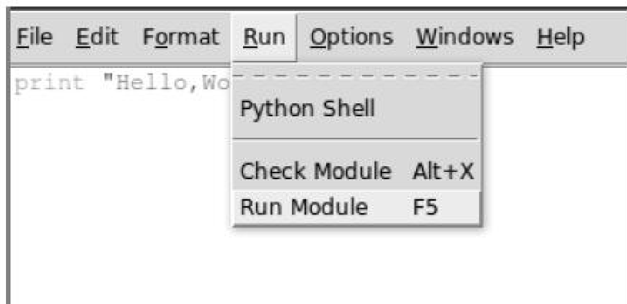


图1-5　单击菜单选项

然后弹出对话框，要求把这个文件保存，将文件保存到一个位置，一定要记住这个位置，并且取个文件名，文件名以.py为扩展名。

都做好之后，单击“确定”按钮，就会发现在另外一个带有“＞＞＞”的界面中，就自动出来了“Hello，World”两个大字。

成功了吗？成功了也别兴奋，因为还没有到庆祝的时候。

在这种情况下，我们依然是在IDE的环境中实现了刚才那段程序的自动执行，如果脱离这个环境呢？

关闭IDLE，打开shell，或者打开cmd，通过命令的方式，进入到你刚才保存的文件目录。

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$
```

这个文件夹里面就一个文件，名为105.py，就是刚才保存的那个文件。

然后在这个shell里面输入：python 105.py。

上面这句话的含义就是：告诉计算机运行一个Python语言编写的程序，程序文件的名称是105.py

我的计算机我做主，于是它乖乖地执行了这条命令。如下图：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python 105.py
Hello,World
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ |
```

还在沉默？可以欢呼了。因为你在程序员道路上迈出了伟大的第二步（思考：什么时候迈出的第一步？）。

1.4.4　解一道题目

题目：请计算19＋2*48/2

读者先自己仰望天空（或者天花板）冥想一下大概如何写，然后继续。

代码：

```
#!/usr/bin/env python
#coding:utf-8

"""

请计算：19+2*48/2

"""

a = 19 + 2 * 4 - 8 / 2
print a
```

提醒初学者，不要复制这段代码，要一个字一个字地敲进去，然后保存（我保存的文件名是：105-1.py）。

在shell或者cmd中，执行：python文件名.py。

执行结果如下图：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python 105-1.py
23
```

好像还是比较简单。

下面对这个简单程序进行解释。

```
#!/usr/bin/env python
```

这一行是必须写的，它能够引导程序找到Python的解释器（或者叫解析器、直译器）。也就是说，不管这个文件保存在什么地方，这个程序都能执行，而不用指定Python的安装路径。

解释器（Interpreter），是一种电脑程序，能够把高级编程语言逐行直接翻译运行。解释器不会一次性把整个程序翻译出来，只像一位“中间人”，每次运行程序时都要先转成另一种语言再运行，因此解释器的程序运行速度比较缓慢。它每翻译一行程序语句就立刻运行，然后再翻译下一行，再运行，如此不停地进行下去。

解释器的好处是它消除了编译整个程序的负担，但也会让运行时的效率打折扣。相对地，编译器并不运行程序或源代码，而是一次将其翻译成另一种语言，如机器码，以供多次运行而无须再编译。其制

成品无须依赖编译器而运行，程序运行速度比较快。（来自《维基百科》）

```
#coding:utf-8
```

这一行是告诉Python，本程序采用的编码格式是utf-8，什么是编码？什么是utf-8？这是一个比较复杂且有历史的问题，此处暂不讨论。只有有了上面这句话，在后面的程序中才能写汉字，否则就会报错。

```
"""
请计算：19+2*4?8/2
"""
```

这一行是让人看的，而计算机看不懂。在Python程序中（别的编程语言也是如此），要写所谓的注释，就是对程序或者某段语句的说明文字，这些文字在计算机执行程序的时候被计算机姑娘忽略，但是，必要的注释又是必不可少的，正如前面说的那样，程序在大多数情况下是给人看的，注释就是帮助人理解程序的。当然，本程序中的注释是不必要的，纯粹是为了说明注释而写。

写注释的方式有两种，一种是单行注释，用“#”开头，另一种是多行注释，用一对“'''”（三个单引号）包裹起来。

用#开头的注释，可以像下面这样来写：

```
#请计算：19+2*48/2
```

这种注释通常写在程序中的某个位置，比如某个语句的前面或者后面。计算机也会忽略这种注释的内容，因为只是给人看的。

一般在程序的开头部分都要写点东西，主要是告诉别人这个程序是用来做什么的，比如刚才的注释，就是说明本程序时要计算那个表达式。

```
a = 19 + 2 * 4 - 8 / 2
```

所谓语句，就是告诉程序要做什么事情。程序是由各种各样的语句组成的。这条语句还有一个名字，叫作赋值语句。19＋2*48/2是一个表达式，最后要计算出一个结果，这个结果就是一个对象。

“＝”，不要理解为数学中的等号，它的作用不是“等于”，而是完成赋值语句中“赋值”的功能。a是变量。这样就完成了一个赋值过程。

语句和表达式的区别：“表达式就是某件事”，“语句是做某件事”。

```
print a
```

这还是一个语句，称之为print语句，就是要打印出a的值（这种说法不是非常严格，但是通常都这么说。严格的说法是打印变量a做对应的对象的值。但这种说法啰嗦，就直接说打印a的值）。

是不是在为看到自己写的第一个程序而欣慰呢？

1.5 字符串

如果要对自然语言分类，则常见的是英语、法语、汉语等，语言学专家还会把他们归类为什么语系。我虽然不是语言学专家，但是也做了一点自己的思考，虽然尚未得到广大人民群众和研究者的广泛认同，但是，我相信那句“真理是掌握在少数人手里的”，至少在这里可以用来给自己壮壮胆。

我对语言的分类方法是：

- 类别一：语言中的两个元素（比如两个字）拼接在一起，出来一个新的元素（比如把“女”和“子”拼接起来，就是“好”，得到了新的字）。
- 类别二：两个元素连接在一起（与“拼接”有差别），也就是这两个元素并列显示。比如“好”和“人”，两个元素连接在一起是“好人”。而3和5拼接（就是整数求和）在一起是8（属于第一类），如果连接，就是35，那就属于第二类了。

上述分类方法也适用于英文，是否适用于其他语种还有待验证。

把我的这种分法抽象一下（因为有这种简单的抽象，才显示出上述语言的分类方法高于一般的语言学专家所认同的方法）：

- 类别一是：△＋□＝○
- 类别二是：△＋□＝△□

根据我个人的研究，在目前知晓的语言范畴中，都没有离开以上两种分类，不是第一类就是第二类。

1.5.1 字符串

在我洋洋自得的时候，我搜索了一下，才发现自己没那么高明，维基百科的“字符串”词条是这么说的：

字符串（String），是由零个或多个字符组成的有限串行。一般记为s＝a[1]a[2]...a[n]。

看到维基百科的伟大了吧，它已经把我所设想的那种情况取了一个形象的名称，叫作字符串，其本质上就是一串字符。

根据这个定义，前面两次让一个程序员感到伟大的“Hello，World”就是一个字符串。或者说不管是用英文还是中文还是别的某种文，写出来的文字都可以作为字符串对待，当然，里面的特殊符号，也可以作为字符串，比如空格等。

在Python中，“万物皆对象”，显然“Hello，World”就是一个对象，这个对象是一个字符串，也就是说，字符串是对象类型，用str表示，这就如同前面遇到的int类型一样。字符串类型的对象通常用单引号或者双引号包裹起来。

```
>>> "I love Python."
'I love Python.'
>>> 'I LOVE PYTHON.'
'I LOVE PYTHON.'
```

从这两个例子中可以看出，不论是使用单引号还是双引号，结果都是一样的。

```
>>> 250
250
>>> type(250)
<type 'int'>
>>> "250"
'250'
>>> type("250")
<type 'str'>
```

在这个例子中同样是250，但区别很大。一个没有放在引号里面，一个放在了引号里面，用type()函数来检验一下，发现它们居然是两种不同的对象类型，前者是int类型，后者则是str类型，即字符串类型。所以，请大家务必注意，不是所有数字都是int（或者float）类型，如果它在引号里面，就是字符串了。如果搞不清楚是什么类型的话，就让type()来帮忙搞定。

操练起来：

```
>>> print "good good study, day day up"
good good study, day day up
>>> print "----good---study---day----up"
----good---study---day----up
```

在print后面打印的都是字符串。注意，引号不是字符串的组成部分，而是在告诉计算机里面包裹着的是一个字符串。如果使用Python 3.x，应该使用print()函数，在Python 3.x中，类似的效果由print()函数完成。

爱思考的读者肯定想到一个问题，如果要把下面这句话看作一个字符串应该怎么做？

```
What's your name?
```

这句话中有一个单引号，如果在交互模式中像上面那样直接输入，就会这样：

```
>>> 'What's your name?'
File "<stdin>", line 1
 'What's your name?'
      ^
SyntaxError: invalid syntax
```

出现了SyntaxError（语法错误）引导的提示，这是在告诉我们这里存在错误，错误的类型就是SyntaxError，后面是对这种错误的解释“invalid syntax”（无效的语法）。特别注意，错误提示的上面，有一个“^”符号，指着一个单引号，是在告诉我们这里出现错误了。

在Python中，这一点是非常友好的，如果语句存在错误，就会将错误输出来，供程序员参考。当然，有时候错误来源比较复杂，需要根据经验和知识进行修改。还有一种修改错误的好办法，就是将错误提示放到Google中进行搜索。

上面那个值的错误原因是什么呢？仔细观察，发现在那句话中事实上有三个单引号，本来一对单引号之间包裹的是一个字符串，现在出现了三个单引号，Computer姑娘迷茫了，她不知道单引号包裹的到底是谁，于是报错。

解决方法一：双引号包裹单引号

```
>>> "What's your name?"
"What's your name?"
```

双引号里面允许出现单引号，反过来，单引号里面也可以包裹双引号，这个可以笼统地称为二者的嵌套。

解决方法二：使用转义符

所谓转义，就是让某个符号不再表示某种含义，而是表示另外一种含义。转义符的作用就是它能够转变符号的含义。在Python中，用“\”（反斜杠）作为转义符（其他很多语言只要有转义符的，都用这个符号）。

```
>>> 'What\'s your name?'
"What's your name?"
```

是不是看到转义符的作用了？

本来单引号不是字符串的一部分，但是如果前面有转义符，那么它就失去了原来的含义，转化为字符串的一部分，相当于一个特殊字符了。

关于转义符的问题后面还会遇到，性子急的读者可以向后翻阅或者自己搜索一下。

1.5.2　变量和字符串

大家对于“变量”已经不陌生了吧，这里是第二次出现了，“一回生二回熟”。一条金科玉律是：在Python中“变量无类型，对象有类型”。变量相当于一个标签，贴在了不同的对象上。这种“贴”的动作，可以通过复制语句完成。

同样，对字符串类型的对象也是这样，能够通过赋值语句，将对象与某个标签（变量）关联起来。

```
>>> b = "hello,world"
>>> b
'hello,world'
>>> print b
hello,world
```

依然请出type()函数，得到变量类型：

```
>>> type(b)
<type 'str'>
```

1.5.3 连接字符串

把两个数字用“+”符号连接起来，比如3+5，结果为8，这其实是求和。但是，对字符串进行类似操作呢？是这样的：

```
>>> "py" + "thon"
'python'
```

两个字符串“相加”，就相当于把两个字符串连接起来。别的运算就别尝试了，没什么意义，肯定报错，不信就试试：

```
>>>"py"-"thon"        # 我这么做，是不是脑袋进水泥了？
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

用“+”号实现连接的确比较简单，不过，有时候你会遇到这样的问题：

```
>>> a = 1989
>>> b = "free"
>>> print b + a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

报错了，其错误原因已经打印出来了（一定要注意看打印出来的信息，这是解决问题的入口）：cannot concatenate'str'and'int'objects。原来a对应的对象是一个int类型的，不能将它和str类型的对象连接起来。怎么办？

用“+”拼接起来的两个对象必须是同一种类型的。如果两个都是数字，毫无疑问是正确的，就是求和；如果都是字符串，那么就得到一个新的字符串。

修改上面的错误，可以通过以下方法：

```
>>> print b + `a`
free1989
```

你是不是照着上面敲过代码呢？你的结果有没有报错？

注意：``是反引号，不是单引号，就是键盘中通常在数字1左边的那个键，在英文半角状态下输入的符号。这种方法，在编程实践中较少应用，特别是在Python 3中，已经把这种方式弃绝了。我想原因就是这个符号太容易和单引号混淆了，且在编程中也不容易看出来，可读性太差。

常言道："困难只有一个，但解决困难的方法不止一种"，既然反引号的可读性不好，在编程实践中就尽量不要使用。于是乎就有了下面的方法，这是被广泛采用的。不仅简单，更主要的是直白，让人一看就懂。

```
>>> print b + str(a)
free1989
```

用str（a）实现将整数对象转换为字符串对象。虽然str是一种对象类型，但是它也能够实现对象类型的转换，这就起到了一个函数的作用。其实前面已经讲过的int也有类似的作用，比如：

```
>>> a = "250"
>>> type(a)
<type 'str'>
>>> b = int(a)
>>> b
250
>>> type(b)
<type 'int'>
```

如果你对int和str比较好奇，可以在交互模式中使用help（int），学习help（str）可以查阅相关的其他资料。

看本书的时候，一定要同时打开计算机，一边看一边操作才不睡觉，尽管本书充满了"水分"，让你难以入睡，但是这种不是小说的书籍，总是在催眠上有很好疗效的。

还有第三种：

```
>>> print b + repr(a)    #repr(a)与上面的类似
free1989
```

这里repr()是一个函数，其实就是反引号的替代品，它能够把结果字符串转化为合法的Python表达式。

可能读者这时候心存疑惑，它们三者之间有区别吗？首先明确，repr()和``是一致的，就不用区别了。接下来需要区别的就是repr()和str，一个最简单的区别：repr是函数，str跟int一样是一种对象类型。不过，仅这么说是不能完全解惑的，幸亏有Google让我辈使用，你会找到很多人对这两者进行区分的内容，我推荐以下这些：

1.When should i use str()and when should i use repr()?

Almost always use str when creating output for end users.

repr is mainly useful for debugging and exploring.For example，if you suspect a string has non printing characters in it，or a float has a small rounding error，repr will show you;str may not.

repr can also be useful for for generating literals to paste into your source code.It can also be used for persistence（with ast.literal_eval or eval），but this is rarely a good idea--if you want editable persisted values，something like JSON or YAML is much better，and if you don't plan to edit them，use pickle.

2.In which cases i can use either of them?

Well，you can use them almost anywhere.You shouldn't generally use them except as described above.

3.What can str()do which repr()can't?

Give you output fit for end-user consumption--not always（e.g.，str（['spam'，'eggs']）isn't likely to be anything you want to put in a GUI），but more often than repr.

4.What can repr()do which str()can't

Give you output that's useful for debugging--again，not always（the default for instances of user-created classes is rarely helpful），but whenever possible.

And sometimes give you output that's a valid Python literal or other expression--but you rarely want to rely on that except for interactive exploration.

以上英文内容来源：

http://stackoverflow.com/questions/19331404/str-vs-repr-functions-in-python-2-7-5。

1.5.4 转义字符

在字符串中，有时需要输入一些特殊的符号，但是，某些符号不能直接输出，就需要用转义符。所谓转义，就是不采用符号本来的含义，而采用另外一种含义。下面列出常用的转义符，如表1-2所示。

表1-2 常用的转义符

转义字符	描　述
\	（在行尾时）续行符
\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格（Backspace）
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy 代表的字符，例如：\o12 代表换行
\xyy	十六进制数，yy 代表的字符，例如：\x0a 代表换行
\other	其他的字符以普通格式输出

以上所有转义符，都可以通过交互模式下print来测试一下，感受实际上是什么样子的。例如：

```
>>> print "hello.I am qiwsir.\                                #这里换行，下一
... My website is 'http://qiwsir.github.io'."
hello.I am qiwsir.My website is 'http://qiwsir.github.io'.
>>> print "you can connect me by qq\\weibo\\gmail"     #\\是为了要后面
you can connect me by qq\weibo\gmail
```

总要自己多练习，才能充分理解转义符的作用。

1.5.5 原始字符串

用转义符能够让字符串中的某些符号表示原来的含义，而不是被解析成某种具有特别能力的符号。为了说话简单，我们常常把那种每个字符都是原始含义的字符串说成原始字符串，比如反斜杠，其不会被看作转义符，就是一个反斜杠。

```
>>> print "I like \npython"
I like
python
```

这里的反斜杠就不是“反斜杠”的原始符号含义，而是和后面的n一起表示换行（转义了）。当然，这似乎没有什么太大影响，但有时候可能会出现问题，比如打印DOS路径。

```
>>> dos = "c:\news"
>>> dos
'c:\news'           #这里貌似没有什么问题
>>> print dos        #当用print来打印这个字符串的时候就出问题了。
c:
ews
```

如何避免？用前面讲过的转义符可以解决，读者试一下。

我当然就不能再用转义符了，要不然就真的“太水”了。我用下面的方法：

```
>>> dos = r"c:\news"
>>> print dos
c:\news
>>> print r"c:\news\python"
c:\news\python
```

状如r"c:\news"，由r开头引起的字符串就是声明了后面引号里的东西是原始字符串，在里面放任何字符都表示该字符的原始含义。

这种方法在做网站设置和网站目录结构的时候非常有用，使用了原始字符串就不需要转义了。

1.5.6　raw_input和print

小孩学说话是一个模仿的过程，周围的人说什么，孩子就重复什

么。如果你已经忘记自己当初是怎么学说话的了，那么就找个小孩子观察一下吧，最好是观察自己的孩子，如果没有，就要抓紧了。

我想用Python实现类似的功能。

在写这个功能前，要了解两个函数：raw_input（如果是使用Python 3.x，请转换为input()）和print。

这两个都是Python的内建函数（built-in function）。关于Python的内建函数，下面都列出来了，供参考。

abs()、divmod()、input()、open()、staticmethod()、all()、enumerate()、int()、ord()、str()、any()、eval()、isinstance()、pow()、sum()、basestring()、execfile()、issubclass()、print()、super()、bin()、file()、iter()、property()、tuple()、bool()、filter()、len()、range()、type()、bytearray()、float()、list()、raw_input()、unichr()、callable()、format()、locals()、reduce()、unicode()、chr()、frozenset()、long()、reload()、vars()、classmethod()、getattr()、map()、repr()、xrange()、cmp()、globals()、max()、reversed()、zip()、compile()、hasattr()、memoryview()、round()、import()、complex()、hash()、min()、set()、apply()、delattr()、help()、next()、setattr()、buffer()、dict()、hex()、object()、slice()、coerce()、dir()、id()、oct()、sorted()、intern()。

这些内建函数，怎么才能知道哪个函数怎么用，且是干什么用的呢？

曾记否？前面使用过的方法在这里再演示一遍，这种方法是学习Python的法宝。

```
>>> help(raw_input)
```

然后就出现：

```
Help on built-in function raw_input in module __builtin__:
raw_input(...)
    raw_input([prompt]) -> string

    Read a string from standard input.  The trailing newline is st
    If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), rai
    On Unix, GNU readline is used if enabled.  The prompt string,
```

```
    is printed without a trailing newline before reading.
```

是不是已经清晰地看到了raw_input()的使用方法了？

还有第二种方法，那就是到Python的官方网站，查看内建函数的说明，网址：https://docs.python.org/2/library/functions.html。

其实，上面列出内建函数名称，就是在这个网页中抄过来的。如果读者愿意跨越发展，就应当用上面的方法把每个内建函数怎么使用、返回值是什么等都查看一遍，做到心中有数。

进入交互模式，操练一番：

```
>>> raw_input("input your name:")
input your name:python
'python'
```

输入名字之后，就返回了输入的内容，用一个变量可以获得这个返回值。

```
>>> name = raw_input("input your name:")
input your name:python
>>> name
'python'
>>> type(name)
<type 'str'>
```

而且，返回的结果是str类型。如果输入的是数字呢？

```
>>> age = raw_input("How old are you?")
How old are you?10
>>> age
'10'
>>> type(age)
<type 'str'>
```

返回的结果仍然是str类型。

再试试print（若不晓得怎么用，可以用help()去看看）。

```
>>> print "hello, world"
hello, world
>>> a = "python"
```

```
>>> b = "good"
>>> print a
python
>>> print a,b
python good
```

比较简单吧。

要特别提醒的是，print的返回值默认是以\n结尾的，所以，每个输出语句之后自动换行。

有了以上两个准备，接下来就可以写一个能够“对话”的小程序了。

```
#!/usr/bin/env python
# coding=utf-8

name = raw_input("What is your name?")
age = raw_input("How old are you?")

print "Your name is:", name
print "You are " + age + " years old."

after_ten = int(age) + 10
print "You will be " + str(after_ten) + " years old after ten year
```

对这段小程序有一些说明。

前面演示了print的使用，除了打印一个字符串之外，还可以打印字符串拼接结果。

```
print "You are " + age + " years old."
```

注意，变量age必须是字符串，如最后的那个语句中：

```
print "You will be " + str(after_ten) + " years old after ten year
```

这句话里面有一个类型转化，将原本是整数型after_ten转化为了str类型，否则就会报错。

同样注意，在after_ten＝int（age）＋10中，通过raw_input得到的是str类型，当age和10求和的时候，需要先用int()函数进行类型转化，才能和后面的整数10相加。

这个小程序基本上把已经学到的东西综合运用了一次。请读者自行调试一下，如果没有通过，则仔细看报错信息，你能够从中获得修改方向的信息。

1.5.7　索引和切片

字符串是一个话题中心，在某些朋友的程序员生涯中，处理字符串的机会可能远远高于处理数字的机会。这可能是因为现在的“程序”越来越多地处理人类的交往信息，所以高级编程语言也越来越多地处理字符串了。

想想字符串的定义，再看看这样一个字符串“python”，还记得前面对字符串的定义吗？它就是几个字符（p、y、t、h、o、n）排列起来。这种排列是非常严格的，不仅仅是字符本身，而且还有顺序，换言之，如果某个字符换了，就变成一个新字符串了；如果这些字符顺序发生了变化，则也将成为一个新字符串。

在Python中，把像字符串这样的对象类型（后面还会冒出来类似的其他有这种特点的对象类型，比如列表）统称为序列。顾名思义，序列就是“有序排列”。

水泊梁山的108个好汉（里面分明也有女的，难道女汉子是从这里来的吗？），就是一个“有序排列”的序列。从老大宋江一直排到第108位金毛犬段景住。在这个序列中，每个人有编号，编号和每个人一一对应：1号是宋江，2号是卢俊义。反过来，通过每个人的姓名，也能找出其对应的编号：武松是多少号？14号。李逵呢？22号。

在Python中，给这些编号取了一个文雅的名字，叫作索引（别的编程语言也这么称呼，不是Python独有的）。

```
>>> lang = "study python"
>>> lang[0]
's'
>>> lang[1]
't'
```

变量lang是贴在字符串“study python”上的标签，如果要得到这个字符串的第一个单词s，可以用lang[0]。

当然，如果你不愿意通过赋值语句让变量lang指向那个字符串，

也可以这样做：

```
>>> "study python"[0]
's'
```

效果是一样的，但是方便程度显而易见。

字符串这个序列的排序方法跟梁山好汉有点不同：第一个不是用数字1表示，而是用数字0表示，其他很多语言也都是从0开始排序的。为什么这样做呢？这就是规定。当然，这个规定是有一定优势的，此处不展开，有兴趣的读者可以去网上搜索一下，有专门对此进行解释的文章。

如表所示，将这个字符串从第一个到最后一个进行了排序，特别注意，两个单词中间的那个空格，也占用了一个位置。

0	1	2	3	4	5	6	7	8	9	10	11
s	t	u	d	y		p	y	t	h	o	n

通过索引能够找到该索引所对应的字符，那么反过来，能不能通过字符找到其在字符串中的索引值呢？怎么找？

```
>>> lang.index("p")
6
```

这样是不是已经能够和梁山好汉的例子对上号了？只不过区别在于程序的第一个索引值是0。

如果某一天，宋江大哥站在大石头上，向各位弟兄大喊："兄弟们，都排好队。"等兄弟们排好之后，宋江说："现在给各位没有老婆的兄弟分配女朋友，我这里已经有了名单，我念到的兄弟站出来，不过我是按照序号来念的。第29号到第34号先出列，到旁边房子等候分配女朋友。"

在前面的例子中lang[1]能够得到原来字符串的第二个字符t，就相当于从原来字符串中把这个"切"出来了。不过，我们这么"切"却不影响原来字符串的完整性，当然也可以理解为将字符t复制一份拿出来了。

类似宋江大哥那样，一次性将几个兄弟一起叫出来，Python也能做到。

```
>>> lang
'study python'        #在前面“切”了若干的字符之后，再看一下该字符串，还是完整
>>> lang[2:9]
'udy pyt'
```

通过lang[2:9]要得到多个（不是一个）字符（来源于原字符串），从返回的结果中可以看出，我们得到的是序号分别对应着2、3、4、5、6、7、8（跟上面的表格对应一下）的字符（包括那个空格）。

不管是得到一个字符还是多个字符，通过索引得到字符的过程都称之为切片。

切片是一个很有意思的东西，可以“切”出不少花样呢。

```
>>> lang
'study python'
>>> b = lang[1:]          #得到从1号到最末尾的字符，这时最后那个不用写
>>> b
'tudy python'
>>> c = lang[:]           #得到所有字符
>>> c
'study python'
>>> d = lang[:10]         #得到从第一个到10号之前的字符
>>> d
'study pyth'
```

在获取切片的时候，如果冒号的前面或者后面的序号不写，则表示两边的某个终点位置，或是开头，或是结尾。也就是，lang[:10]的效果和lang[0:10]是一样的。

```
>>> e = lang[0:10]
>>> e
'study pyth'
```

那么，lang[1:]和lang[1:11]效果一样吗？请思考后作答。

```
>>> lang[1:11]
'tudy pytho'
>>> lang[1:]
'tudy python'
```

答案是：不一样，你思考对了吗？

在“切”字符的时候，如果冒号后面有数字，所得到的切片不包含该数字所对应的字符（前包括，后不包括）。那么，是不是可以这样呢？lang[1:12]不包括12号（事实上没有12号），是不是可以得到1号到11号对应的字符呢？

```
>>> lang[1:12]
'tudy python'
>>> lang[1:13]
'tudy python'
```

果然结果和猜测的一样，即如果第二个数字大于字符串的长度，得到的返回结果就自动到最大长度位置终止。但是请注意，这种获得切片的做法在编程实践中是不提倡的。特别是如果后面要用到循环的时候，这样做很可能会遇到麻烦。

如果在“切片”的时候，冒号左右都不写数字，就是前面所操作的c＝lang[:]，其结果是变量c的值与原字符串一样，即“复制”了一份。注意，这里的“复制”打上了引号，意思是如同复制，是不是真的复制呢？可以用下面的方式检验一下：

```
>>> id(c)
3071934536L
>>> id(lang)
3071934536L
```

id()的作用还记得吗？

从上面可以看出，两个内存地址一样，说明c和lang两个变量指向的是同一个对象。用c＝lang[:]的方式并没有生成一个新的字符串，而是将变量c这个标签也贴在了原来那个字符串上了。

```
>>> lang = "study python"
>>> c = lang
```

如果这样操作，变量c和lang是不是指向同一个对象呢？读者可以自行检验。

1.5.8　基本操作

所有序列都有如下基本操作，字符串是序列的子集。

- len()：返回序列长度。
- ＋：连接两个序列。
- *：重复序列元素。
- in：判断元素是否存在于序列中。
- max()：返回最大值。
- min()：返回最小值。
- cmp（str1，str2）：比较两个序列值是否相同。

通过下面的例子，将这几个基本操作在字符串上的使用演示一下。

1）“＋”连接字符串

```
>>> str1 = 'abcd'
>>> str2 = 'abcde'
>>> str1 + str2
'abcdabcde'
>>> str1 + "-->" + str2
'abcd-->abcde'
```

不要小看“＋”号，到此只是学了字符串这一种序列，后面还会遇到列表、元组两种序列，都能够如此实现拼接。

2）in

```
>>> "a" in str1
True
>>> "de" in str1
False
>>> "de" in str2
```

in用来判断某个字符串是不是在另外一个字符串内，或者判断某个字符串内是否包含某个字符串，如果包含，就返回True，否则返回False。

3）最值

```
>>> max(str1)
'd'
>>> max(str2)
'e'
>>> min(str1)
'a'
```

在一个字符串中，每个字符在计算机内都是有编码的，也就是对应着一个数字，min()和max()就是根据这些数字获得最小值和最大值，然后对应出相应的字符。关于这种编号是多少，可以搜索有关字符编码或者ASCII编码，很容易查到。

4）比较

```
>>> cmp(str1, str2)
-1
```

将两个字符串进行比较，首先将字符串中的符号转化为对应的数字（怎么对应数字了？请参照ASCII理解），然后再比较。如果返回的数值小于零，说明第一个小于第二个；等于0，则两个数值相等；大于0，则第一个数值大于第二个数值。为了能够明白其所以然，进入下面的分析。

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord(' ')
32
```

ord()是一个内建函数，能够返回某个字符（注意，是一个字符，而不是多个字符组成的串）所对应的ASCII值（是十进制的），字符a在ASCII中的值是97，空格在ASCII中也有值，是32。反过来，根据整数值得到相应字符，可以使用chr()：

```
>>> chr(97)
'a'
>>> chr(98)
'b'
```

于是，得到如下比较结果：

```
>>> cmp("a", "b")              #a-->97，b-->98，97小于98，所以a小于b
-1
>>> cmp("abc", "aaa")
1
>>> cmp("a", "a")
0
```

看看下面的比较是怎么进行的呢？

```
>>> cmp("ad", "c")
-1
```

在字符串的比较中，两个字符串的第一个字符先比较，如果相等，就比较下一个，如果不相等，就返回结果。如果直到最后还相等，就返回0。位数不够时，按照“没有”处理（注意，“没有”不是0，0在ASCII中对应的是NUL），位数多的那个大。ad中的a先和后面的c进行比较，显然a小于c，于是返回结果-1。但进行下面的比较，是最容易让人迷茫的。读者能不能根据刚才阐述的比较原理理解得到的结果呢？

```
>>> cmp("123", "23")
-1
>>> cmp(123,23)          #也可以比较整数，这时候就是整数的直接比较了。
1
```

5）“*”

字符串中的“乘法”含义是重复那个字符串，在某些时候很好用的，比如要打印一个华丽的分割线：

```
>>> str1 * 3
'abcdabcdabcd'
>>> print "-" * 20          #不用输入很多个`-`
--------------------
```

6）len()

要知道一个字符串有多少个字符，一种方法是从头开始盯着屏幕数。哦，这不是计算机在干活，是“键客”在干活。

键客，不是剑客。剑客是以剑为武器的侠客；而键客是以键盘为武器的侠客。

计算机这样来数字符串长度：

```
>>> a="hello"
>>> len(a)
5
```

函数len()返回该字符串长度。

```
>>> m = len(a)          #把结果返回后赋值给一个变量
>>> m
5
>>> type(m)            #这个返回值（变量）是一个整数型
<type 'int'>
```

1.5.9 常用的字符串方法

字符串的方法有很多，可以通过dir来查看：

```
>>> dir(str)
    ['__add__', '__class__', '__contains__', '__delattr__', '__doc
```

这么多字符串方法当然不用都介绍，因为有一种方法，读者可以随用随查阅每个字符串方法是如何使用的。

```
>>> help(str.isalpha)
Help on method_descriptor:
isalpha(...)
    S.isalpha() -> bool
    Return True if all characters in S are alphabetic
    and there is at least one character in S, False otherwise.
```

按照这里的说明，在交互模式下进行实验。

```
>>> "python".isalpha()           #字符串全是字母，应该返回True
True
>>> "2python".isalpha()          #字符串含非字母，返回False
False
```

以下仅列举几个常用的方法。

1）split()

其作用是将字符串根据某个分割符进行分割。

```
>>> a = "I LOVE PYTHON"
>>> a.split(" ")
['I', 'LOVE', 'PYTHON']
```

这里用空格作为分割，得到一个名字叫作列表（list）的返回值，关于列表的内容，后续会介绍。还能用别的分隔吗？

```
>>> b = "www.itdiffer.com"
>>> b.split(".")
['www', 'itdiffer', 'com']
```

2）去掉字符串两头的空格

比如让用户输入一些信息，有的用户有时候会在信息（比如，自己的名字就是字符串）前面或者后面加空格。这些空格是没用的，在使用所输入的信息时，必须要把这些空格去掉。

Python中去掉空格的方法有如下几种：

```
S.strip()：去掉字符串的左右空格
S.lstrip()：去掉字符串的左边空格
S.rstrip()：去掉字符串的右边空格
```

例如：

```
>>> b=" hello "     #两边有空格
>>> b.strip()
'hello'
>>> b
' hello '
```

特别注意，原来的值没有变化，而是新返回了一个结果。

```
>>> b.lstrip()       #去掉左边的空格
'hello '
>>> b.rstrip()       #去掉右边的空格
' hello'
```

3）字符大小写的转换

英文有时候要用到大小写转换。最有名的是驼峰命名，里面就有一些大写和小写。如果有兴趣，可以来这里学习自动将字符串转化为驼峰命名形式的方法（参见：https://github.com/qiwsir/algorithm/blob/master/string_to_hump.md）。相关的方法有：

- S.upper()

- S.lower()
- S.capitalize()
- S.isupper()
- S.islower()
- S.istitle()

看例子：

```
>>> a = "qiwsir,python"
>>> a.upper()                  #将小写字母完全变成大写字母
'QIWSIR,PYTHON'
>>> a                          #原数据对象并没有改变
'qiwsir,python'
>>> b = a.upper()
>>> b
'QIWSIR,PYTHON'
>>> c = b.lower()              #将所有的大写字母变成小写字母
>>> c
'qiwsir,python'

>>> a
'qiwsir,python'
>>> a.capitalize()             #把字符串的第一个字母变成大写
'Qiwsir,python'
>>> a                          #原数据对象没有改变
'qiwsir,python'
>>> b = a.capitalize()
>>> b
'Qiwsir,python'

>>> a = "qiwsir,github"
>>> a.istitle()
False
>>> a = "QIWSIR"               #当全是大写的时候，返回False
>>> a.istitle()
False
>>> a = "qIWSIR"
>>> a.istitle()
False
>>> a = "Qiwsir,github"        #如果这样，也返回False
>>> a.istitle()
False
>>> a = "Qiwsir"               #这样是True
>>> a.istitle()
True
>>> a = 'Qiwsir,Github'        #这样也是True
>>> a.istitle()
True
```

```
>>> a = "Qiwsir"
>>> a.isupper()
False
>>> a.upper().isupper()
True
>>> a.islower()
False
>>> a.lower().islower()
True
>>> a = "This is a Book"
>>> a.istitle()
False
>>> b = a.title()       #这样就把所有单词的第一个字母转化为大写
>>> b
'This Is A Book'
>>> b.istitle()         #判断每个单词的第一个字母是否为大写
```

4）join连接字符串

用“+”能够连接字符串，但不是什么情况下都能够如愿。比如，将列表（列表是另外一种类型）中的每个字符（串）元素拼接成一个字符串，并且用某个符号连接，但如果用“+”会比较麻烦。用字符串的join方法就比较容易实现。

```
>>> b
'www.itdiffer.com'
>>> c = b.split(".")
>>> c
['www', 'itdiffer', 'com']
>>> ".".join(c)
'www.itdiffer.com'
>>> "*".join(c)
'www*itdiffer*com'
```

1.5.10　字符串格式化输出

什么是格式化？在维基百科中有专门的词条，是这么说的：

格式化是指对磁盘或磁盘中的分区（Partition）进行初始化的一种操作，这种操作通常会导致现有的磁盘或分区中所有的文件被清除。

不知道你是否知道这种“格式化”。显然，此格式化非我们这里所说的，我们说的是字符串的格式化，或者说是“格式化字符串”，表示的意思就是：

格式化字符串，是C、C＋＋等程序设计语言printf类函数中用于指定输出参数的格式与相对位置的字符串参数。其中的转换说明（conversion specification）用于把随后对应的0个或多个函数参数转换为相应的格式输出；格式化字符串中转换说明以外的其他字符原样输出。

这也是来自维基百科的定义。在这个定义中，用C语言作为例子，并且用了其输出函数来说明。在Python中，也有同样的操作和类似的函数print，此前我们已经了解一二了。

将那个定义说得通俗一些：字符串格式化就是要先制定一个模板，在这个模板中某个或者某几个地方留出空位来，然后在那些空位填上字符串。那么，那些空位需要用一个符号来表示，这个符号通常被叫作占位符（仅仅是占据着那个位置，并不是输出的内容）。

```
>>> "I like %s"
'I like %s'
```

在这个字符串中，有一个符号“%s”，这是一个占位符，可以被其他的字符串代替。比如：

```
>>> "I like %s" % "python"
'I like python'
>>> "I like %s" % "Pascal"
'I like Pascal'
```

这是较为常用的一种字符串输出方式。

不同的占位符，表示那个位置应该被不同类型的对象填充，如表1-3所示。常用的只有%s、%d和%f，如果需要其他的，到这里来查即可。

看例子：

```
>>> a = "%d years" % 15
>>> print a
15 years
```

表1-3　占位符

占　位　符	说　　明
%s	字符串（采用 str()的显示）
%r	字符串（采用 repr()的显示）
%c	单个字符
%b	二进制整数
%d	十进制整数
%e	指数（底数为 e）
%f	浮点数

当然，还可以在一个字符串中设置多个占位符，就像下面一样：

```
>>> print "Suzhou is more than %d years. %s lives in here." % (250
Suzhou is more than 2500 years. qiwsir lives in here.
```

对于浮点数字的打印输出，还可以限定输出的小数位数和其他样式：

```
>>> print "Today's temperature is %.2f" % 12.235
Today's temperature is 12.23
>>> print "Today's temperature is %+.2f" % 12.235
Today's temperature is +12.23
```

注意：在上面的例子中，没有实现四舍五入的操作，只是截取，但是上面的例子也的确太特殊了。如果读者有兴趣，可以换一个数，自己试试，在一般情况下是能够实现四舍五入的。

关于类似的操作还有很多变化，比如输出格式的宽度是多少等。如果读者在编程中遇到了，可以到网上查找。在这里给一个参考图示，也是从网上下载的，如图1-6所示。

数字	格式	输出	描述
3.1415926	{:.2f}	3.14	保留小数点后两位
3.1415926	{:+.2f}	+3.14	带符号保留小数点后两位
-1	{:+.2f}	-1.00	带符号保留小数点后两位
2.71828	{:.0f}	3	不带小数
5	{:0>2d}	05	数字补零 (填充左边, 宽度为2)
5	{:x<4d}	5xxx	数字补x (填充右边, 宽度为4)
10	{:x<4d}	10xx	数字补x (填充右边, 宽度为4)
1000000	{:,}	1,000,000	以逗号分隔的数字格式
0.25	{:.2%}	25.00%	百分比格式
1000000000	{:.2e}	1.00e+09	指数记法
13	{:10d}	13	右对齐 (默认, 宽度为10)
13	{:<10d}	13	左对齐 (宽度为10)
13	{:^10d}	13	中间对齐 (宽度为10)

图1-6　字符串格式

其实，上面这种格式化方法，常常被认为太“古老”了。因为在Python中还有新的格式化方法。

```
>>> s1 = "I like {}".format("python")
>>> s1
'I like python'
>>> s2 = "Suzhou is more than {0} years. {1} lives in here.".forma
>>> s2
'Suzhou is more than 2500 years. qiwsir lives in here.'
```

这就是Python非常提倡的string.format()的格式化方法，其中{}作为占位符。

这种方法真得非常好，而且非常简单，只需要将对应的东西按照顺序在format后面的括号中排列好，分别对应占位符{}即可。

如果你觉得还不明确，还可以这样来做。

```
>>> print "Suzhou is more than {year} years. {name} lives in here.
Suzhou is more than 2500 years. qiwsir lives in here.
```

真的很简洁、优雅。

还有一种格式化的方法是“字典格式化”，这里仅仅举一个例子，如果读者要了解“字典”的含义，本教程后续会有的。

```
>>> lang = "python"
>>> print "I love %(program)s" % {"program":lang}
I love python
```

这里列举了三种基本格式化的方法，你喜欢那种？我推荐：string.format()。

1.6 字符编码

在Python 2.x中，字符编码是一个让人困惑的问题，这个问题在Python 3.x中自然解决了。由此可以说，未来是Python 3的。但是，由于前面已经分析过的原因，在一段时间内Python 2.x还不能完全丢弃，甚至不少工程项目还是以它为主。所以，还要将字符编码问题单独叙述。

如果一个字符串都是英文，就没有所谓编码问题。但在我们的环境中，中文是我们不得不用的。

```
>>> name = '老齐'
>>> name
'\xe8\x80\x81\xe9\xbd\x90'
```

你在交互模式中遇到过上面的情形吗？这就是显示汉字的问题，英文就不这样了。

难道这是中文的错吗？看来投胎真的是一个技术活。是的，投胎是技术活，但上面的问题不是中文的错。

1.6.1 编码

什么是编码？这是一个比较玄乎的问题，也不好下一个普通定义。我看到有的教材中有定义，且不敢说其定义不对，但至少是不容易理解。

“古代打仗，击鼓进攻、鸣金收兵”这就是编码。把要传达给士兵的命令对应为一定的其他形式，比如命令“进攻”，经过信息传递，如图1-7所示。

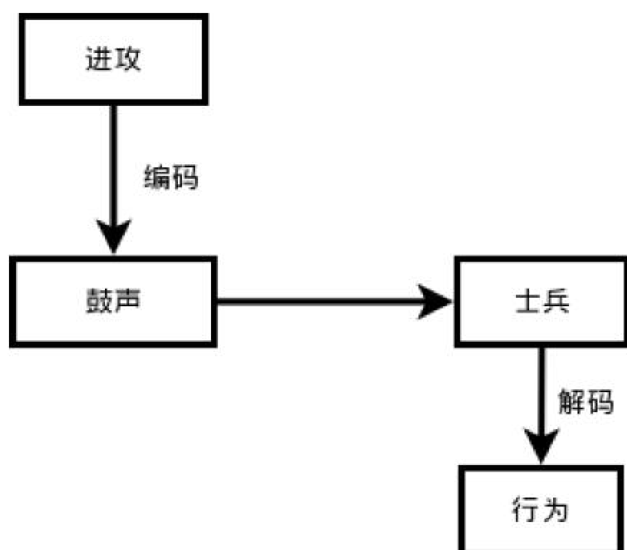


图1-7　信息传递

1）长官下达进攻命令，传令员将这个命令编码为鼓声。

2）鼓声在空气中传播，比传令员的嗓子吼出来的声音传播得更远，士兵听到后也不会有歧义，这就是“进攻”命令被编码成鼓声之后的优势所在。

3）士兵听到鼓声，就是接收到信息，如果接受过训练或者有人告诉过他们，他们就知道这是命令进攻，这个过程就是解码。所以，编码方案要有两套：一套在信息发出者那里，另外一套在信息接受者这里。经过解码之后，士兵明白了才行动。

以上过程比较简单，但真实的编码和解码过程比这个复杂。不过，原理都差不多。

举一个似乎遥远，其实不久前人们都在使用的东西做例子：电报（以下引用的内容来自《维基百科》）。

电报是通信业务的一种，在19世纪初发明，是最早使用电进行通信的方法。电报大为加快了消息的流通，是工业社会的一项重要发明。早期的电报只能在陆地上通信，后来使用了海底电缆，开展了越洋服务。到了20世纪初，开始使用无线电波发电报，电报业务基本上已能抵达地球上大部分地区。电报主要用作传递文字讯息，使用电报

技术用作传送图片称为传真。

中国出现首条电报线路是1871年，由英国、俄国及丹麦敷设，从中国香港经上海至日本长崎，且是海底电缆。由于清政府的反对，电缆被禁止在上海登录。后来丹麦公司不理清政府的禁令，将线路引至上海公共租界，并在1871年6月3日起开始收发电报。至于中国首条自主敷设的线路，是由福建巡抚丁日昌在中国台湾所建，1877年10月完工，连接台南及高雄。1879年，北洋大臣李鸿章在天津、大沽及北塘之间架设电报线路，用作军事通信。1880年，李鸿章奏准开办电报总局，由盛宣怀任总办。并在1881年12月开通天津至上海的电报服务。李鸿章説："五年来，我国创设沿江沿海各省电线，总计一万多里，国家所费无多，巨款来自民间。当时正值法人挑衅，将帅报告军情，朝廷传达指示，均相机而动，无丝毫阻碍。中国自古用兵，从未如此神速。出使大臣往来问答，朝发夕至，相隔万里好似同居庭院。举设电报一举三得，既防止外敌侵略，又加强国防，亦有利于商务。"天津官电局于庚子遭乱全毁。1887年，台湾巡抚刘铭传敷设了福州至台湾的海底电缆，是中国首条海底电缆。1884年，北京电报开始建设，采用"安设双线，由通州展至京城，以一端引入署中，专递官信，以一端择地安置用便商民"，8月5日，电报线路开始建设，所有电线杆一律漆成红色。8月22日，位于北京崇文门外大街西的喜鹊胡同的外城商用电报局开业。同年8月30日，位于崇文门内泡子和以西的吕公堂开局，专门收发官方电报。

为了传达汉字，电报部门准备由4位数字或3位罗马字构成的代码，即中文电码，采用发送前将汉字改写成电码发出，收电报后再将电码改写成汉字的方法。

注意：这里出现了电报中用的"中文电码"，这就是一种编码，将汉字对应成阿拉伯数字，从而能够用电报发送汉字。

1873年，法国驻华人员威基杰参照《康熙字典》的部首排列方法，挑选了常用汉字6800多个，编成了第一部汉字电码本《电报新书》。

电报中的编码被称为摩尔斯电码，英文是Morse Code。

摩尔斯电码（英语：Morse Code）是一种时通时断的信号代码，通过不同的排列顺序来表达不同的英文字母、数字和标点符号。是由美国人萨缪尔·摩尔斯在1836年发明。

摩尔斯电码是一种早期的数字化通信形式，但是它不同于现代只

使用0和1两种状态的二进制代码，它的代码包括五种：点（.）、划（-）、每个字符间短的停顿（在点和划之间的停顿）、每个词之间中等的停顿以及句子之间长的停顿。

看来电报员是一个技术活，不同长短的停顿都代表了不同意思。哦，对了，有一个老片子叫《永不消逝的电波》，保证你看完之后才知道，里面根本就没有讲电报是怎么编码的。

摩尔斯电码在海事通信中被作为国际标准一直使用到1999年。1997年，当法国海军停止使用摩尔斯电码时，发送的最后一条消息是："所有人注意，这是我们在永远沉寂之前最后的一声呐喊！"

```
****-/*----/----*/****-/****-/*----/---**/*----/****-/*----/-****/***-
```

```
****-/*----/----*/****-/****-/*----/---**/*----/****-/*----/-****/***-
```

我瞪着眼看了老长时间，这两行不是一样的吗？

不管这个了，总之，这就是编码。

1.6.2　计算机中的字符编码

抄一段维基百科对字符编码的解释：

字符编码（英语：Character Encoding），也称为字集码，是把字符集中的字符编码为指定集合中某一对象（例如：比特模式、自然数串行、8位组或者电脉冲），以便文本在计算机中存储和通过通信网络的传递。常见的例子包括将拉丁字母表编码成摩斯电码和ASCII。其中，ASCII将字母、数字和其他符号编号，并用7比特的二进制来表示这个整数。通常会额外使用一个扩充的比特，以便于以1个字节的方式存储。

在计算机技术发展的早期，如ASCII（1963年）和EBCDIC（1964年）这样的字符集逐渐成为标准。但这些字符集的局限很快就变得明显，于是人们开发了许多方法来扩展它们。对于支持包括东亚CJK字符家族在内的写作系统的要求能支持更大量的字符，并且需要一种系统而不是临时的方法实现这些字符的编码。

在这个世界上，有好多不同的字符编码。但是，它们不是自己随

便搞搞的，而是要有一定的基础，往往是以名叫ASCII的编码为基础。

ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统。它主要用于显示现代英语，而其扩展版本EASCII则可以部分支持其他西欧语言，并等同于国际标准ISO/IEC 646。由于万维网使得ASCII广为通用，直到2007年12月，逐渐被Unicode取代。

上面的引文中已经说了，现在我们用的编码标准已经变成Unicode了（Python3.x就是用了Unicode），那么什么是Unicode呢？还是抄一段来自维基百科的说明：

Unicode（中文：万国码、国际码、统一码、单一码）是计算机科学领域里的一项业界标准。它对世界上大部分的文字系统进行了整理、编码，使得电脑可以用更为简单的方式来呈现和处理文字。

Unicode伴随着通用字符集的标准而发展，同时也以书本的形式对外发表。Unicode至今仍在不断增修，每个新版本都加入更多新的字符。目前最新的版本为7.0.0，已收入超过十万个字符（第十万个字符在2005年获采纳）。Unicode涵盖的数据除了视觉上的字形、编码方法、标准的字符编码外，还包含了字符特性，如大小写字母。

听这名字：万国码，那就一定包含了中文。但是，光有一个Unicode还是不够用（可以访问《维基百科》网站查看相关说明），还要有其他的一些编码实现方式，Unicode的实现方式称为Unicode转换格式（Unicode Transformation Format，简称为UTF），于是乎有了一个我们在很多时候都会看到的utf-8。

什么是utf-8？还是看维基百科上怎么说的吧：

UTF-8（8-bit Unicode Transformation Format）是一种针对Unicode的可变长度字符编码，也是一种前缀码。它可以用来表示Unicode标准中的任何字符，且其编码中的第一个字节仍与ASCII兼容，这使得原来处理ASCII字符的软件不需要或只做少部份修改，即可继续使用。因此，它逐渐成为电子邮件、网页及其他存储或发送文字的应用中优先采用的编码。

是不是理解了呢？前面写程序的时候，曾经出现过coding:utf-8的字样，就是在告诉Python我们要用什么字符编码。

1.6.3 encode和decode

encode()和decode()是两个内置函数。

codecs.encode（obj[，encoding[，errors]]）:Encodes obj using the codec registered for encoding.

codecs.decode（obj[，encoding[，errors]]）:Decodes obj using the codec registered for encoding.

Python2默认的编码是ASCII，通过encode()可以将对象的编码转换为指定编码格式（称作“编码”），而decode是这个过程的逆过程（称作“解码”）。

做一个实验，才能理解：

```
>>> a = "中"
>>> type(a)
<type 'str'>
>>> a
'\xe4\xb8\xad'
>>> len(a)
3

>>> b = a.decode()
>>> b
u'\u4e2d'
>>> type(b)
<type 'unicode'>
>>> len(b)
1
```

在做这个实验之前，或许还不是很迷茫（知道得越多越迷茫），实验做完了，自己也迷茫了。别急躁，对编码问题的理解要慢慢来，如果一时理解不了，就先按照要求做，做着做着就豁然开朗了。

变量a引用了一个字符串类型对象，但严格地讲是字节串，因为它是经过编码后的字节组成的序列。也就是你在上面的实验中看到的“中”这个字在计算机中编码之后的字节表示。（关于字节可以搜索一下）。用len（a）来度量它的长度，它是由三个字节组成的。

然后通过decode函数将字节串转变为字符串，并且这个字符串是按照Unicode编码的。在Unicode编码中，一个汉字对应一个字符，这

时候度量它的长度就是1。

反过来，一个Unicode编码的字符串也可以转换为字节串。

```
>>> c = b.encode('utf-8')
>>> c
'\xe4\xb8\xad'
>>> type(c)
<type 'str'>
>>> c == a
```

关于编码问题先到这里点到为止吧。因为再扯，还会扯出问题来，读者肯定感到不满意，因为还没有知其所以然。

1.6.4　避免中文是乱码

“避免中文是乱码”是一个具有很强操作性的问题。

首先，提倡使用utf-8编码方案，因为它跨平台不错。

经验一，在开头声明：

```
# -*- coding: utf-8 -*-
```

有朋友问我“-*-”有什么作用，那个就是为了好看，爱美之心人皆有，更何况程序员？当然，也可以写成：

```
# coding:utf-8
```

经验二，遇到字符（节）串，立刻转化为unicode，不要用str()，直接使用unicode()：

```
unicode_str = unicode('中文', encoding='utf-8')
print unicode_str.encode('utf-8')
```

经验三，如果对文件操作，打开文件的时候，最好用codecs.open替代open（关于文件的操作，请参阅后续内容。）

```
import codecs
codecs.open('filename', encoding='utf8')
```

最后，如果用Python3，这种编码的烦恼会少一点。

1.7 列表

此前，已经知道了三种Python的对象类型：int、float和str。

这一节中的list类型，也是Python的一种对象类型，翻译为：列表。下面的加粗字，请读者注意：

list在Python中具有非常强大的功能。

1.7.1 定义

在Python中，用方括号表示一个list：[]

方括号里面的元素类型，可以是int，也可以是str类型的数据，甚至也能够是True/False这种布尔值。看下面的例子，要特别注意阅读注释。

```
>>> a=[]                  #定义了一个空的列表，变量a相当于一个贴在其上的标签
>>> type(a)
<type 'list'>             #用内置函数type()查看变量a引用对象的类型，为list
>>> bool(a)               #用内置函数bool()看看a的布尔值，因为是空的，所以为Fal
False
>>> print a               #打印
[]
```

bool()是一个布尔函数，在后续章节会详述。它的作用就是来判断一个对象是“真”还是“假”（空）。如果像上面的例子那样，列表中什么也没有就是空的，用bool()函数来判断，得到False，从而显示它是空的。

不能总玩“空”的，来点“实”的吧。

```
>>> a=['2', 3, 'qiwsir.github.io']
>>> a
['2', 3, 'qiwsir.github.io']
>>> type(a)
<type 'list'>
>>> bool(a)
```

```
True
>>> print a
['2', 3, 'qiwsir.github.io']
```

一个列表中能够容纳多少东西？“有容乃大”是对列表最好的形容了，它的大小仅受制于硬件设备和你的意愿。

如果你已经了解了别的语言，比如比较常见的Java，里面有一个跟list相似的数据类型——数组——但是两者还是有区别的。在Java中，数组中的元素必须是基本数据类型中的某一个，也就是要么都是int类型，要么都是char类型等，不能一个数组中既有int类型又有char类型。这是因为Java中的数组需要提前声明，声明的时候就确定了里面元素的类型。但是尽管Python中的list与Java中的数组有类似的地方——都是[]包裹的——list中的元素是任意类型的，可以是int、str，还可以是list，甚至是dict等。所以，有一句话说：列表是Python中的苦力，什么都可以干。

1.7.2 索引和切片

关于索引和切片的含义在字符串章节已经熟知了。所以，这里可以很快用起来。

```
>>> url = "qiwsir.github.io"
>>> url[2]
'w'
>>> url[:4]
'qiws'
>>> url[3:9]
'sir.gi'
```

在列表中也有类似的操作。只不过是以元素为单位，而不是以字符为单位进行索引。

```
>>> a
['2', 3, 'qiwsir.github.io']
>>> a[0]                        #索引序号也是从0开始
'2'
>>> a[1]
3
>>> a[:2]
['2', 3]
>>> a[1:]
```

```
[3, 'qiwsir.github.io']
```

列表和字符串两种类型都属于序列（都是一些对象按照某个次序排列起来，这是序列的最大特征），因此，他们有很多类似的地方。如刚才演示的索引和切片是非常一致的。

```
>>> lang = "python"
>>> lang.index("y")
1
>>> lst = ['python','java','c++']
>>> lst.index('java')
1
```

索引数字从左边开始编号，第一个是0，然后依次增加1。

此外，还有一种编号方式是从右边开始，右边第一个可以编号为-1，然后向左依次是：-2，-3，...，依次类推下来。这对字符串、列表等各种序列类型都适用。

```
>>> lang
'python'
>>> lang[-1]
'n'
>>> lst
['python', 'java', 'c++']
>>> lst[-1]
'c++'
```

从右边开始编号，第-1号是右边第一个。但是，如果要切片的话，应该注意：

```
>>> lang[-1:-3]
''
>>> lang[-3:-1]
'ho'
>>> lst[-3:-1]
['python', 'java']
```

序列的切片，一定要左边的数字小于右边的数字，lang[-1:-3]就没有遵守这个规则，返回的是一个空。

1.7.3 反转

反转在编程中常常会用到。通过举例来说明反转的方法：

```
>>> alst = [1,2,3,4,5,6]
>>> alst[::-1]
[6, 5, 4, 3, 2, 1]
>>> alst
[1, 2, 3, 4, 5, 6]
```

当然，对于字符串也可以：

```
>>> lang
'python'
>>> lang[::-1]
'nohtyp'
>>> lang
'python'
```

是否注意到，不管是str还是lst，反转之后原来的值没有改变。这就说明，这里的反转，不是在“原地”把原来的值倒过来，而是新生成了一个值，生成的值跟原来的值相比，是倒过来了。

这是一种非常简单的方法，虽然我在写程序的时候常常使用，但并不是十分推荐，因为它有时候让人感觉迷茫。Python还有另外一种方法让list反转，且比较容易理解和阅读，特别推荐之：

```
>>> list(reversed(alst))
[6, 5, 4, 3, 2, 1]
```

这个比较简单，而且很容易看懂，不是吗？

顺便给出reversed函数的详细说明：

```
>>> help(reversed)
Help on class reversed in module __builtin__:

class reversed(object)
 |  reversed(sequence) -> reverse iterator over values of the sequ
 |
 |  Return a reverse iterator
```

它返回一个可以迭代的对象（关于迭代的问题，请参阅后续内容），不过已经将原来的序列对象反转了。比如：

```

```

```
>>> list(reversed("abcd"))
['d', 'c', 'b', 'a']
```

很好、很强大，特别推荐使用。

1.7.4 对list的操作

在字符串那部分已经提到过，所有的序列都有几种基本操作。列表是一种序列，当然如此。

1.len()

```
>>> lst
['python', 'java', 'c++']
>>> len(lst)
3
```

2. +，连接两个序列

```
>>> lst
['python', 'java', 'c++']
>>> alst
[1, 2, 3, 4, 5, 6]
>>> lst + alst
['python', 'java', 'c++', 1, 2, 3, 4, 5, 6]
```

3.*，重复元素

```
>>> lst
['python', 'java', 'c++']
>>> lst * 3
['python', 'java', 'c++', 'python', 'java', 'c++', 'python', 'java
```

4.in

列表lst还是前面的值：

```
>>> "python" in lst
True
>>> "c#" in lst
False
```

5.max()和min()

以int类型元素为例：

```
>>> alst
[1, 2, 3, 4, 5, 6]
>>> max(alst)
6
>>> min(alst)
1
>>> max(lst)
'python'
>>> min(lst)
'c++'
```

6.cmp()

采用上面的方法，进行比较：

```
>>> lsta = [2,3]
>>> lstb = [2,4]
>>> cmp(lsta,lstb)
-1
>>> lstc = [2]
>>> cmp(lsta,lstc)
1
>>> lstd = ['2','3']
>>> cmp(lsta,lstd)
-1
```

7.追加元素

```
>>> a = ["good","python","I"]
>>> a
['good', 'python', 'I']
>>> a.append("like")        #向list中添加str类型"like"
>>> a
['good', 'python', 'I', 'like']
```

```
>>> a.append(100)             #向list中添加int类型100
>>> a
['good', 'python', 'I', 'like', 100]
```

官方文档这样描述list.append()方法：

```
list.append(x)
Add an item to the end of the list; equivalent to a[len(a):] = [x]
```

是否已经理解了list.append（x）的含义呢？即将新的元素x追加到list的尾部。

如果注意看上面官方文档中的那句话，应该注意到后面半句：equivalent to a[len（a）:]＝[x]，意思是说list.append（x）等效于a[len（a）:]＝[x]。这也相当于告诉我们另外一种追加元素的方法，并且两种方法等效。

```
>>> a
['good', 'python', 'I', 'like', 100]
>>> a[len(a):]=[3]        #len(a),即得到list的长度，这个长度是指list中的元
>>> a
['good', 'python', 'I', 'like', 100, 3]
>>> len(a)
6
>>> a[6:]=['xxoo']
>>> a
['good', 'python', 'I', 'like', 100, 3, 'xxoo']
```

1.7.5　列表的函数

什么是方法？方法和函数有什么区别？这里暂不区分和解释。请把这个名词“方法”抽象出来，等到后面自然明了（下面的内容中，没有区分“函数”和“方法”，读者不要介意这个名词）。

列表是Python中的苦力，那么它都有哪些函数呢？或者对它能做什么呢？

```
>>> dir(list)
    ['__add__', '__class__', '__contains__', '__delattr__', '__del
```

上面的结果中，以双下画线开始和结尾的暂时不管，如

add（以后会管的）。就剩下以下几个了：

```
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', '
```

下面对这些函数进行说明和演示，这都是在编程实践中常常要用到的。

1.append和extend

前面提到的列表基本操作中有list.append（x），也就是将某个元素x追加到已知的一个列表的尾部。

除了将元素追加到列表中，还能够将两个列表合并，或者说将一个列表追加到另外一个列表中。按照惯例，首先还是看官方文档中的描述：

```
list.extend(L)
Extend the list by appending all the items in the given list; equi
```

向所有正在阅读本书的朋友提供一个成为优秀程序员的必备：看官方文档。

将官方文档的这句话翻译过来：

通过将所有元素追加到已知list来扩充它，相当于a[len（a）:]＝L

英语太烂，翻译太差。直接看例子，更明白：

```
>>> la
[1, 2, 3]
>>> lb
['qiwsir', 'python']
>>> la.extend(lb)
>>> la
[1, 2, 3, 'qiwsir', 'python']
>>> lb
['qiwsir', 'python']
```

上面的例子显示：有两个list，一个是la，另外一个是lb，将lb这个列表extend到la的后面，也就是把lb中的所有元素加入到la中，即

让la扩容。

学程序一定要有好奇心，我在交互环境中经常实验自己的想法，有时甚至是比较愚蠢的想法。

```
>>> la = [1,2,3]
>>> b = "abc"
>>> la.extend(b)
>>> la
[1, 2, 3, 'a', 'b', 'c']
>>> c = 5
>>> la.extend(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: 'int' object is not iterable
```

从上面的实验中，你能够有什么心得？原来，如果操作extend（str）的时候，str被以字符为单位拆开，然后追加到la里面。

如果extend的对象是数值型，则报错。

所以，extend的对象是一个list，如果是str，则Python会先把它按照字符为单位转化为list再追加到已知list。

别忘记了前面官方文档的后半句话，它的意思是：

```
>>> la
[1, 2, 3, 'a', 'b', 'c']
>>> lb
['qiwsir', 'python']
>>> la[len(la):]=lb
>>> la
[1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']
```

list.extend（L）等效于list[len（list）:]＝L，L是待并入的list。

概括起来，extend函数也是将另外的元素增加到一个已知列表中，元素必须是iterable，什么是iterable？

iterable，中文含义是“可迭代的”。在Python中还有一个词，就是iterator，这个叫作“迭代器”，这两者有着区别和联系。

为了解释iterable（可迭代的），又引入了一个词“迭代”，什么是迭代呢？

尽管我们很多文档是用英文写的，但是，如果你能充分利用汉语来理解某些名词，将是非常有帮助的。因为在汉语中，不仅仅表音，而且能从词语组合中体会到该术语的含义。比如“激光”，这是汉语。英语是从“light amplification by stimulated emission of radiation”出来的“laser”，它是一个造出来的词，因为此前人们不知道在那种条件下发出来的是什么。但是汉语不然，反正用一个“光”就可以概括了，只不过这个“光”不是传统概念中的“光”，而是由于“受激”辐射得到的光，故名“激光”。是不是汉语很牛？

“迭”在汉语中的意思是“屡次，反复”，如高潮迭起。跟“代”组合就可以理解为“反复‘代’”，是不是有点“子子孙孙”的意思了？“结婚-生子-子成长-结婚-生子-子成长-……”，你是不是也在这个“迭代”的过程中呢？

给个稍微严格的定义，来自维基百科：“迭代是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。”

某些类型的对象是“可迭代”（iterable）的，但如何判断一个对象是不是可迭代的？下面演示一种方法（事实上还有别的方式）：

```
>>> astr = "python"
>>> hasattr(astr,'__iter__')
False
```

这里用内建函数hasattr()判断一个字符串是否是可迭代的，返回了False。用同样的方式可以判断：

```
>>> alst = [1,2]
>>> hasattr(alst,'__iter__')
True
>>> hasattr(3, '__iter__')
False
```

hasattr()的判断本质就是看那个类型中是否有__iter__函数。读者可以用dir()找一找，在数字、字符串、列表中谁有__iter__函数。同样还可找一找dict、tuple两种类型对象是否含有这个方法。

以上穿插了一个新的概念“iterable”（可迭代的），现在回到extend上，这个函数需要的参数就是iterable类型的对象。

```
>>> new = [1, 2, 3]
>>> lst = ['python','qiwsir']
```

```
>>> lst.extend(new)
>>> lst
['python', 'qiwsir', 1, 2, 3]
>>> new
[1, 2, 3]
```

通过extend函数，将[1，2，3]中的每个元素都拿出来，然后塞到lst里面，从而得到一个跟原来的对象元素不一样的列表，比原来的多了三个元素。上面说得有点啰嗦，只不过是为了把过程完整表达出来。

从上面的演示中可以看出，lst经过extend函数操作之后，变成了一个貌似“新”的列表。

```
>>> new = [1,2,3]
>>> id(new)
3072383244L

>>> lst = ['python', 'qiwsir']
>>> id(lst)
3069501420L
```

用id()能够看到两个列表分别在内存中的“窝”的编号。

```
>>> lst.extend(new)
>>> lst
['python', 'qiwsir', 1, 2, 3]
>>> id(lst)
3069501420L
```

虽然lst经过extend()方法之后比原来扩容了，但是，并没有离开原来的“窝”，即在内存中还是“旧”的，只不过里面的内容增多了。相当于两口之家，经过一番云雨之后，又增加了一个小宝宝，那么这个家是“新”的还是“旧”的呢？角度不同或许说法不一。

这就是列表的一个重要特征：列表是可以修改的。这种修改，不是复制一个新的，而是在原地进行修改。

其实，append()对列表的操作也是如此，不妨用同样的方式看看。

说明：虽然这里的lst内容和上面的一样，但是，重新在shell中输入id会变化。也就是内存分配的“窝”的编号变了。

```
>>> lst = ['python','qiwsir']
>>> id(lst)
3069501388L
>>> lst.append(new)
>>> lst
['python', 'qiwsir', [1, 2, 3]]
>>> id(lst)
3069501388L
```

显然，append()也是原地修改列表。

如果对于extend()提供的不是iterable类型对象，会如何呢？

```
>>> lst.extend("itdiffer")
>>> lst
['python', 'qiwsir', 'i', 't', 'd', 'i', 'f', 'f', 'e', 'r']
```

它把一个字符串"itdiffer"转化为['i'，'t'，'d'，'i'，'f'，'f'，'e'，'r']，然后将这个列表作为参数，提供给extend，并将列表中的元素塞入原来的列表中。

```
>>> num_lst = [1,2,3]
>>> num_lst.extend(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

这就报错了。错误提示告诉我们，那个数字8是int类型的对象，不是iterable的。

这里讲述两个让列表扩容的函数append()和extend()，总结其共同点：

- 都是原地修改列表。
- 既然是原地修改，就不返回值。

原地修改没有返回值，就不能赋值给某个变量。

```
>>> one = ["good", "good", "study"]
>>> another = one.extend(["day","day","up"])      #没有返回值
>>> another                                        #这样的，什么也没有得
>>> one
['good', 'good', 'study', 'day', 'day', 'up']
```

那么两者有什么不一样呢？再看下面的例子之前，读者能不能通过前面的操作总结一下？请敲代码尝试，然后看下面的例子：

```
>>> lst = [1,2,3]
>>> lst.append(["qiwsir","github"])
>>> lst
[1, 2, 3, ['qiwsir', 'github']]       #append的结果
>>> len(lst)
4

>>> lst2 = [1,2,3]
>>> lst2.extend(["qiwsir","github"])
>>> lst2
[1, 2, 3, 'qiwsir', 'github']       #extend的结果
>>> len(lst2)
5
```

append是整建制地追加，extend是个体化扩编。

2.count

count的作用是数一数某个元素在该list中出现多少次，也就是某个元素有多少个。官方文档是这么说的：

```
list.count(x)
Return the number of times x appears in the list.
```

一定要不断实验才能理解文档中精炼的表达。

```
>>> la = [1, 2, 1, 1, 3]
>>> la.count(1)
3
>>> la.append('a')
>>> la.append('a')
>>> la
[1, 2, 1, 1, 3, 'a', 'a']
>>> la.count('a')
2
>>> la.count(5)       #la中没有5，但是不报错，返回的是数字0
0
```

3.index

前面已经讲过，不赘述，但是为了完整，也占个位置吧。

```
>>> la
[1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']
>>> la.index(3)
2
>>> la.index('qi')        #如果不存在，就报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: 'qi' is not in list
```

依然是上一条官方解释，主要目的是熟悉看文档：

```
list.index(x)
Return the index in the list of the first item whose value is x. I
```

是不是说得非常清楚明白了？如果没有搞清楚，赶快学英语吧。未来，如果在写代码这个职业上发展，英语是绕不开的。

4.insert

除了向列表中追加元素，在现实中，还应该有“插入”。Python提供了这样的操作，list.insert（i，x）就是向列表插入元素的函数。

如果学会了看官方文档，则学习任何语言都是小菜一碟了。继续看Python官方文档是如何解释insert()的：

```
list.insert(i, x)
Insert an item at a given position. The first argument is the inde
```

根据官方文档的说明，我们做下面的实验，请读者从实验中理解：

```
>>> all_users
['qiwsir', 'github', 'io']
>>> all_users.insert("python")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: insert() takes exactly 2 arguments (1 given)
```

请注意看报错的提示信息，insert()应该供给两个参数，但是这里只给了一个，所以报错没商量啦。

```
>>> all_users.insert(0, "python")
>>> all_users
['python', 'qiwsir', 'github', 'io']

>>> all_users.insert(1, "http://")
>>> all_users
['python', 'http://', 'qiwsir', 'github', 'io']
```

list.insert（i，x）中的i是将元素x插入到列表中的位置，即将x插入到索引值是i的元素前面。注意，索引是从0开始的。

有一种操作挺有意思的，如下：

```
>>> length = len(all_users)
>>> length
5
>>> all_users.insert(length, "algorithm")
>>> all_users
['python', 'http://', 'qiwsir', 'github', 'io', 'algorithm']
```

在原来的all_users中，最大索引值是4，如果要all_users.insert（5，"algorithm"），则表示将"algorithm"插入到索引值是5的前面，但是没有。换个说法，5前面就是4的后面。所以，就是追加了。

其实，还可以这样：

```
>>> a = [1,2,3]
>>> a.insert(9, 777)
>>> a
[1, 2, 3, 777]
```

也就是说，如果遇到那个i已经超过了最大索引值，会自动将所要插入的元素放到列表的尾部，即追加。

5.pop和remove

对列表，不仅能增加元素，还能被删除之。删除元素的方法有两个，它们分别是：

- list.remove（x）

Remove the first item from the list whose value is x.It is an error if there is no such item.

- list.pop（[i]）

Remove the item at the given position in the list，and return it.If no index is specified，a.pop()removes and returns the last item in the list.（The square brackets around the i in the method signature denote that the parameter is optional，not that you should type square brackets at that position.You will see this notation frequently in the Python Library Reference.）

我学习Python有一个习惯，就是用学习物理的方法来学习。不知道读者的物理学得怎么样？如果当初没有学好也不用担心，跟着我的思路走，不仅能学好Python，还能学好物理。物理中有一个非常重要的研究和学习方法：先实验，然后总结规律。

对于这种方法，读者回味一下前面的内容，是不是隐隐然有所感受呢？

先实验list.remove（x），这是一个能够删除list元素的方法，同时上面引用的官方说明告诉我们，如果x没有在list中，就会报错。

```
>>> all_users
['python', 'http://', 'qiwsir', 'github', 'io', 'algorithm']
>>> all_users.remove("http://")
>>> all_users
['python', 'qiwsir', 'github', 'io', 'algorithm']

>>> all_users.remove("tianchao")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list

>>> lst = ["python", "java", "python", "c"]
>>> lst.remove("python")
>>> lst
['java', 'python', 'c']
```

在第三段实验中，列表内有两个'python'字符串，当删除后，发现结果只删除了第一个'python'字符串，第二个还在。原因何在？请仔细看前面的文档说明：remove the first item...。

注意：

- 如果正确删除，不会有任何反馈。没有消息就是好消息，因为是对列表进行原地修改。
- 如果所删除的对象不在list中，就报错。注意阅读报错信息：x not in list。

读者在阅读的时候如果没有关注某些细节，往往就失去了本书的某些精华，比如我在前面的很多操作中，都使用了一个名为lst的变量，而不是用list，为什么呢？因为list是Python的保留字。

什么是保留字？在Python中，某些词语或者拼写是不能被用户拿来做变量、函数、类等命名，因为它们已经被语言本身先占用了。这些就是所谓的保留字。在Python中，以下是保留字，不能用于变成任何命名。

and，assert，break，class，continue，def，del，elif，else，except，exec，finally，for，from，global，if，import，in，is，lambda，not，or，pass，print，raise，return，try，while，with，yield

这些保留字都是我们在编程中要用到的，有的你已经在前面遇到了。

如果能够在执行删除之前，先判断列表中是否有那个对象，有了再删，没有就别费事了，是不是更显出程序的智能程度高呢？

这的确是一个不错的想法。你觉得应该如何实现？

```
>>> all_users
['python', 'qiwsir', 'github', 'io', 'algorithm']
>>> "python" in all_users                    # 用in来判断是否在list中
True

>>> if "python" in all_users:
...     all_users.remove("python")
...     print all_users
... else:
...     print "'python' is not in all_users"
...
['qiwsir', 'github', 'io', 'algorithm']         # 删除了"python"元素

>>> if "python" in all_users:
...     all_users.remove("python")
...     print all_users
... else:
```

```
...     print "'python' is not in all_users"
...
'python' is not in all_users                    # 因为已经删除了，所以
```

上述代码，就是两段小程序，我是在交互模式中运行的，相当于小实验。这里其实用了一个后面才会讲到的东西：if-else语句。不过，我觉得即使没有学习，你也能看懂，因为它非常接近自然语言了。

精力旺盛的读者还可以将上面的问题编写成一段小程序单独运行。

接着看另外一个删除list.pop（[i]）的程序。方法还是：看看文档，做做实验。

```
>>> all_users
['qiwsir', 'github', 'io', 'algorithm']
>>> all_users.pop()         #list.pop([i]),圆括号里面是[i]，表示这个序号是
'algorithm'                              #默认删除最后一个，并且将该结果返回

>>> all_users
['qiwsir', 'github', 'io']

>>> all_users.pop(1)        #指定删除编号为1的元素"github"
'github'

>>> all_users
['qiwsir', 'io']
>>> all_users.pop()
'io'

>>> all_users               #只有一个元素了，该元素编号是0
['qiwsir']
>>> all_users.pop(1)        #但是非要删除编号为1的元素，结果报错。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

list.remove（x）中的参数是列表中元素，即删除某个元素；list.pop（[i]）中的i是列表中元素的索引值，这个i用方括号包裹起来，意味着还可以不写任何索引值，如上面的操作结果，就是删除列表的最后一个。

留一个思考题，如果要像前面那样，能不能事先判断一下要删除的编号是不是在list的长度范围（用len（list）获取长度）以内，然后

进行删除或者不删除操作？

6.reverse

reverse比较简单，就是把列表的元素顺序反过来。

```
>>> a = [3,5,1,6]
>>> a.reverse()
>>> a
[6, 1, 5, 3]
```

注意，是原地反过来，不是另外生成一个新的列表，所以，它没有返回值。跟这个类似的有一个内建函数reversed，建议读者了解一下这个函数的使用方法。

因为list.reverse()不返回值，所以不能实现对列表的反向迭代，如果要这么做，可以使用reversed函数。

7.sort

sort是对列表进行排序。文档中是这么写的：

```
sort(...)
L.sort(cmp=None, key=None, reverse=False) -- stable sort IN PLACE;

>>> a = [6, 1, 5, 3]
>>> a.sort()
>>> a
[1, 3, 5, 6]
```

list.sort()也是让列表进行原地修改，没有返回值。默认情况如上面的操作，实现的是从小到大的排序。

```
>>> a.sort(reverse=True)
>>> a
[6, 5, 3, 1]
```

这样，实现了从大到小的排序。

在前面的函数说明中，还有一个参数key，这个怎么用呢？不知道你是否熟悉电子表格，能够设置按照某个关键字进行排序。Python

当然不会比电子表格弱。

```
>>> lst = ["python","java","c","pascal","basic"]
>>> lst.sort(key=len)
>>> lst
['c', 'java', 'basic', 'python', 'pascal']
```

这是以字符串的长度为关键词进行排序的。

对于排序，还有一个更为常用的内建函数sorted，读者可以去研究一番，并且比较一下两种排序的各自特点。

顺便指出，排序是一个非常有研究价值的话题，不仅仅是这一个函数。有兴趣的读者可以去网上搜一下与排序相关的知识。

1.8 比较列表和字符串

列表和字符串两种对象类型有不少相似的地方，也有很大的区别。有必要对它们做个简要的比较，以便能深刻理解，此外也是复习，“温故而知新”。

1.8.1 相同点

两者都属于序列类型，由此，那些属于序列的操作对两者都适用。

在对序列类型理解的基础上，用趋于专业的语言描述：它的每一个元素都可以通过指定一个编号（行话叫做“偏移量”或者“索引值”）的方式得到，而要想一次得到多个元素，可以使用切片。偏移量或者索引值从0开始，到总元素数减1结束。

```
>>> welcome_str = "Welcome you"
>>> welcome_str[0]
'W'
>>> welcome_str[len(welcome_str) - 1]
'u'
>>> welcome_str[:4]
'Welc'
>>> a = "python"
>>> a * 3
'pythonpythonpython'

>>> git_list = ["qiwsir","github","io"]
>>> git_list[0]
'qiwsir'
>>> git_list[len(git_list) - 1]
'io'
>>> git_list[0:2]
['qiwsir', 'github']
>>> b = ['qiwsir']
>>> b * 7
['qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiws
```

还有一些操作是类似的：

```
>>> first = "hello,world"
>>> welcome_str
'Welcome you'
>>> first+","+welcome_str
'hello,world,Welcome you'

>>> language = ['python']
>>> git_list
['qiwsir', 'github', 'io']
>>> language + git_list
['python', 'qiwsir', 'github', 'io']

>>> len(welcome_str)
11
>>> len(git_list)
3
```

其他关于序列的基本操作，此处不再重复。

1.8.2 区别

列表和字符串的最大区别是：列表是可以改变，字符串不可变。

首先看对列表的这些操作，其特点是在原处将列表进行了修改：

```
>>> git_list
['qiwsir', 'github', 'io']

>>> git_list.append("python")
>>> git_list
['qiwsir', 'github', 'io', 'python']

>>> git_list[1]
'github'
>>> git_list[1] = 'github.com'
>>> git_list
['qiwsir', 'github.com', 'io', 'python']

>>> git_list.insert(1, "algorithm")
>>> git_list
['qiwsir', 'algorithm', 'github.com', 'io', 'python']

>>> git_list.pop()
'python'
```

```
>>> del git_list[1]
>>> git_list
['qiwsir', 'github.com', 'io']
```

以上这些操作，如果用在str上都会报错，比如：

```
>>> welcome_str
'Welcome you'

>>> welcome_str[1]='E'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> del welcome_str[1]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion

>>> welcome_str.append("E")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

如果要修改一个str，不得不这样。

```
>>> welcome_str
'Welcome you'
>>> welcome_str[0]+"E"+welcome_str[2:]      #重新生成一个str
'WElcome you'
>>> welcome_str                             #对原来的没有任何影响
'Welcome you'
```

其实，在这种做法中就是重新生成了一个str。

1.8.3 多维列表

在字符串中，每个元素只能是字符，在列表中，元素可以是任何类型。前面见到的大多是数字或者字符，其实还可以这样：

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix[0][1]
```

```
2
>>> mult = [[1,2,3],['a','b','c'],'d','e']
>>> mult
[[1, 2, 3], ['a', 'b', 'c'], 'd', 'e']
>>> mult[1][1]
'b'
>>> mult[2]
'd'
```

以上显示了多维列表以及访问方式。在多维的情况下，里面的列表被当成一个元素对待。

1.8.4　列表和字符串的互相转化

以下涉及split()和join()两个函数，在前面字符串部分已经见过。一回生，二回熟，特别是在已经学习了列表的基础上见面，应该有更深刻的理解。

str.split()

这个内置函数实现的是将str转化为list。其中str＝""是分隔符。

请先在交互模式下做如下操作：

```
>>>help(str.split)
```

得到了对这个内置函数的完整说明。特别强调：这是一种非常好的学习方法（本书的特色就是教给读者一些方法，所谓“授人以鱼不如授人以渔”）。

split（...）S.split（[sep[，maxsplit]]）->list of strings

Return a list of the words in the string S，using sep as the delimiter string.If maxsplit is given，at most maxsplit splits are done.If sep is not specified or is None，any whitespace string is a separator and empty strings are removed from the result.

不管是否看懂上面这段话，都来看一下例子。

```
>>> line = "Hello.I am qiwsir.Welcome you."
>>> line.split(".")                #以英文的句点为分隔符
```

```
['Hello', 'I am qiwsir', 'Welcome you', '']

#这个1,就是表达了官方文档中的：If maxsplit is given, at most maxsplit s
>>> line.split(".",1)
['Hello', 'I am qiwsir.Welcome you.']

>>> name = "Albert Ainstain"            #也有可能用空格作为分隔符
>>> name.split(" ")
['Albert', 'Ainstain']
```

下面的例子，让你更有点惊奇了。

```
>>> s = "I am, writing\npython\tbook on line"
#这个字符串中有空格、逗号、换行\n、tab缩进\t 符号

>>> print s
I am, writing
python  book on line
>>> s.split()                    #用split(),但是括号中不输入任何参数
['I', 'am,', 'writing', 'python', 'book', 'on', 'line']
```

如果split()不输入任何参数，显示就是见到任何分割符号，就用其分割了。

1.8.5 "[sep]".join (list)

join可以说是split的逆运算，举例：

```
>>> name
['Albert', 'Ainstain']
>>> "".join(name)
'AlbertAinstain'
>>> ".".join(name)
'Albert.Ainstain'
>>> " ".join(name)
'Albert Ainstain'
```

回到上面那个神奇的例子中，可以这样使用join.：

```
>>> s = "I am, writing\npython\tbook on line"
>>> print s
I am, writing
python  book on line
```

```
>>> s.split()
['I', 'am,', 'writing', 'python', 'book', 'on', 'line']
>>> " ".join(s.split())           #重新连接，不过有一点遗憾，am后面逗号还
                                  #怎么去掉？
'I am, writing python book on line'
```

1.9 元组

1.9.1 定义

先看一个例子：

```
>>> #变量引用str
>>> s = "abc"
>>> s
'abc'

>>>#如果这样写，就会是...
>>> t = 123, 'abc', ["come","here"]
>>> t
(123, 'abc', ['come', 'here'])
```

上面例子中并没有报错，也没有“最后一个有效”，而是将对象作为一个新类型：tuple（元组），赋值给了变量t。

元组是用圆括号括起来的，元素之间用逗号隔开。（特别提醒，不管是圆括号还是逗号，都是英文半角的。这个并非杞人忧天，不知道什么原因，很多敲代码的朋友居然把自己的电脑默认输入方式为中文，结果常常在代码中插入全角符号。）

元素中的元素可以是任何Python对象类型。

其实，你不应该对元组陌生，前面讲述字符串的格式化输出时，有这样一种方式：

```
>>> print "I love %s, and I am a %s" % ('python', 'programmer')
I love python, and I am a programmer
```

这里的圆括号就是一个元组。

元组也是一种序列，这一点与列表、字符串类似。它的特点就是其中的元素不能更改，这一点与列表不同，倒是跟字符串类似；它的元素又可以是任何类型的数据，这一点与列表相同，但不同于字符串。

```
>>> t = 1,"23",[123,"abc"],("python","learn")          #元素多样性，近
>>> t
(1, '23', [123, 'abc'], ('python', 'learn'))

>>> t[0] = 8                                            #不能原地修改，近str
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> t.append("no")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

从上面的简单比较似乎可以认为，元组就是一个融合了部分列表和部分字符串属性的杂交产物。

1.9.2 索引和切片

前面有了关于列表和字符串的知识基础，它们都是序列类型，元组也是。因此，元组的基本操作和它们是一样的。

例如：

```
>>> t
(1, '23', [123, 'abc'], ('python', 'learn'))
>>> t[2]
[123, 'abc']
>>> t[1:]
('23', [123, 'abc'], ('python', 'learn'))

>>> t[2][0]                     #还能这样呀，哦，对了，list中也能这样
123
>>> t[3][1]
'learn'
```

关于序列的基本操作在元组上的表现就不一一展示了，读者自行调试吧。

但是这里要特别提醒，如果一个元组中只有一个元素，应该在该元素后面加一个半角的英文逗号。

```
>>> a = (3)
```

```
>>> type(a)
<type 'int'>

>>> b = (3,)
>>> type(b)
<type 'tuple'>
```

如果不加那个逗号就不是元组，加了才是，这也是为了避免让Python误解你要表达的内容。

列表和元组之间可以实现转化，分别使用list()和tuple()实现。

```
>>> t
(1, '23', [123, 'abc'], ('python', 'learn'))
>>> tls = list(t)                                   #tuple-->list
>>> tls
[1, '23', [123, 'abc'], ('python', 'learn')]

>>> t_tuple = tuple(tls)                            #list-->tuple
>>> t_tuple
(1, '23', [123, 'abc'], ('python', 'learn'))
```

1.9.3 用途

既然元组是列表和字符串的杂合，那么它有什么用途呢？不是用列表和字符串就可以了吗？

有些情况只需要列表和字符串，但是，世界是复杂的，我们要解决的问题不全是简单问题，就如同自然语言一样，虽然有的词汇看似可有可无，用别的也能替换之，但我们依然要在某些情况下使用它们。

一般认为元组有这些特点，并且也是它使用的情景：

- 元组比列表操作速度快。如果定义了一个值的常量集，并且唯一要用它做的是不断地遍历（遍历是一种操作，读者可以看后面的for循环）它，请使用元组代替列表。
- 如果对不需要修改的数据进行“写保护”，可以使代码更安全，这时使用元组而不是列表。如果必须要改变这些值，则需要执行元组到列表的转换。
- 元组可以在字典（另外一种对象类型，请参考后面的内容）中

被用作key，但是列表不行。因为字典的key必须是不可变的，元组本身是不可改变的。

- 元组可以用在字符串格式化中。

1.10 字典

你现在还用字典吗？随着网络的发展，用字典的人越来越少了，不少人习惯于在网上搜索。在很久以前，我曾拥有一本小小的《新华字典》。

《新华字典》是中国第一部现代汉语字典，最早的名字叫《伍记小字典》，但未能编纂完成。自1953年，开始重编，其凡例完全采用《伍记小字典》。从1953年开始出版，经过反复修订，但是以1957年商务印书馆出版的《新华字典》作为第一版。由原新华辞书社编写，1956年并入中科院语言研究所（现中国社科院语言研究所）词典编辑室，新华字典由商务印书馆出版。历经几代上百名专家学者10余次大规模修订，重印200多次。成为迄今为止世界出版史上发行量最高的字典。

在这里讲到字典，不是为了回忆，而是提醒读者想想我们如何使用字典：先查索引，然后通过索引找到相应内容，不用从头开始一页一页地找，这种方法能够快捷地直达目标。

正是基于这种需要，Python中有了一种叫作dictionary的对象类型，翻译过来就是“字典”，用dict表示。

假设有一种需要，要存储城市和电话区号，苏州的区号是0512，唐山的是0315，北京的是011，上海的是012。用前面已经学习过的知识，可以这样来做：

```
>>> citys = ["suzhou", "tangshan", "beijing", "shanghai"]
>>> city_codes = ["0512", "0315", "011", "012"]
```

用一个列表来存储城市名称，然后用另外一个列表一一对应地保存区号。假如要输出苏州的区号，可以这样做：

```
>>> print "{0} : {1}".format(citys[0], city_codes[0])
suzhou : 0512
```

在city_codes中表示区号的元素没有用整数型，而是使用了字符串类型，你知道为什么吗？如果用整数，就是这样的：

```
>>> suzhou_code = 0512
>>> print suzhou_code
330
```

怎么会这样？原来，在Python中如果按照上面那样做，0512被认为是一个八进制的数，用print打印的时候，将它转换为了十进制输出。关于进制转换问题，可以在网上搜索一下有关资料，此处不详述。一般是用几个内建函数实现：int()，bin()，oct()，hex()。

用两个列表分别来存储城市和区号，似乎能够解决问题。但是，这不是最好的选择，因为Python还提供了另外一种方案，那就是“字典”。

1.10.1 创建字典

方法1：

创建一个空的字典，然后可以加入东西。

```
>>> mydict = {}
>>> mydict
{}
```

不要小看“空”，在编程中，“空”是很重要。

当然可以创建一个不空的字典：

```
>>> person = {"name":"qiwsir", "site":"qiwsir.github.io", "languag
>>> person
{'name': 'qiwsir', 'language': 'python', 'site': 'qiwsir.github.io
```

"name":"qiwsir"有一个优雅的名字：键值对。前面的name叫做键（key），后面的qiwsir是前面的键所对应的值（value）。在一个字典中，键是唯一的，不能重复。值则对应于键，且值可以重复。键值之间用英文的冒号，每一对键值之间用英文的逗号隔开。

```
>>> person['name2']="qiwsir"          #增加键值对的方法
>>> person
{'name2': 'qiwsir', 'name': 'qiwsir', 'language': 'python', 'site'
```

用这样的方法可以向一个字典中增加“键值对”，那么，增加了值之后，那个字典对象还是原来的内存地址吗？即也要探讨字典是否能原地修改？（列表可以，因为列表是可变的；字符串和元组都不行，因为它们是不可变的）。

```
>>> ad = {}
>>> id(ad)
3072770636L
>>> ad["name"] = "qiwsir"
>>> ad
{'name': 'qiwsir'}
>>> id(ad)
3072770636L
```

实验表明，字典可以原地修改，即它是可变的。

方法2：

利用元组建构字典，方法如下：

```
>>> name = (["first", "Google"], ["second", "Yahoo"])
>>> website = dict(name)
>>> website
{'second': 'Yahoo', 'first': 'Google'}
```

或者用这样的方法：

```
>>> ad = dict(name="qiwsir", age=42)
>>> ad
{'age': 42, 'name': 'qiwsir'}
```

方法3：

这个方法，跟以上方法的不同在于使用fromkeys：

```
>>> website = {}.fromkeys(("third","forth"),"facebook")
>>> website
{'forth': 'facebook', 'third': 'facebook'}
```

特别注意，字典中的“键”，必须是不可变对象；“值”可以是任意类型的对象。

```
>>> dd = {(1,2):1}
```

```
>>> dd
{(1, 2): 1}
>>> dd = {[1,2]:1}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

1.10.2 访问字典的值

字典类型的对象是以键值对的形式存储数据的，所以，只要知道键，就能得到值，这在本质上就是一种映射关系。

映射，就好比“物体”和“影子”的关系，“形影相吊”，两者之间是映射关系。此外，映射也是一个严格的数学概念：A是非空集合。A到B的映射是指：A中每个元素都对应到B中的某个元素。

既然是映射，就可以通过字典的“键”找到相应的“值”。

```
>>> person
{'name2': 'qiwsir', 'name': 'qiwsir', 'language': 'python', 'site'
>>> person['name']
'qiwsir'
>>> person['language']
'python'
```

“键”很关键，因为通过“键”能够增加“值”，通过“键”能够改变“值”，通过“键”也能够访问到“值”。

本小节开头的城市和区号的关系，也可以用字典来存储和读取。

```
>>> city_code = {"suzhou":"0512", "tangshan":"0315", "beijing":"01
>>> print city_code["suzhou"]
```

既然字典是键值对的映射，就不用考虑所谓“排序”问题了，只要通过键就能找到值，至于这个键值对的位置在哪里就不用考虑了。比如，刚才建立的city_code。

```
city_code。
>>> city_code
{'suzhou': '0512', 'beijing': '011', 'shanghai': '012', 'tangshan'
```

虽然这里显示的和刚刚赋值的时候顺序有别，但是不影响读取其中的值。

在列表中，通过索引值可以得到某个元素。那么在字典中有索引吗？当然没有，因为它没有顺序，又哪里来的索引呢？所以，在字典中就不要什么索引和切片了。

字典中的这类以“键值对”的映射方式存储数据是一种非常高效的方法，比如要读取值的时候，如果用列表，Python需要从头开始读，直到找到指定的那个索引值。但是，在字典中是通过“键”来得到值，要高效得多。正是这个特点，“键值对”这样的形式可以用来存储大规模的数据，因为检索快捷，规模越大越明显。所以，mongdb这种非关系型数据库在大数据方面比较流行。

1.10.3　基本操作

列举字典的基本操作：

- len（d），返回字典（d）中的键值对的数量。
- d[key]，返回字典（d）中的键（key）的值。
- d[key]=value，将值（value）赋给字典（d）中的键（key）。
- del d[key]，删除字典（d）的键（key）项（将该键值对删除）。
- key in d，检查字典（d）中是否含有键为key的项。

依次进行演示。

```
>>> city_code
{'suzhou': '0512', 'beijing': '011', 'shanghai': '012', 'tangshan'
>>> len(city_code)
4
```

以city_code为操作对象，len（city_code）的值是4，表明有四组键值对，也可以说是四项。如果增加项目，则：

```
>>> city_code["nanjing"] = "025"
>>> city_code
{'suzhou': '0512', 'beijing': '011', 'shanghai': '012', 'tangshan'
```

突然发现北京的区号写错了。可以这样修改，这进一步说明字典是可变的。

```
>>> city_code["beijing"] = "010"
>>> city_code
{'suzhou': '0512', 'beijing': '010', 'shanghai': '012', 'tangshan'
```

不仅北京的写错了，上海的也写错了，干脆删除，使用del，将那一项全部删掉。

```
>>> city_code["shanghai"]
'012'
>>> del city_code["shanghai"]
>>> city_code["shanghai"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'shanghai'
```

"shanghai"的那个键值对项已经删除了，所以不能找到，用in来看看，返回的是False。

```
>>> "shanghai" in city_code
False
```

1.10.4 字符串格式化输出

字符串格式化输出问题是前面已经存在的内容，这里再次提到是因为用字典也可以实现格式化字符串的目的。

```
>>> city_code = {"suzhou":"0512", "tangshan":"0315", "hangzhou":"0
>>> " Suzhou is a beautiful city, its area code is %(suzhou)s" % c
' Suzhou is a beautiful city, its area code is 0512'
```

这种写法非常简洁，而且很有意思，有人说它很酷。

其实，更酷的是下面的——模板

在做网页开发的时候通常要用到模板，你只需要写好HTML代码，然后将某些部位空出来，等着Python后台提供相应的数据即可。当然，下面所演示的是玩具代码，基本没有什么实用价值，因为在真

实的网站开发中，这样的知识很少用。但是，它绝非花拳绣腿，而是你能够明了其本质，至少了解到一种格式化方法的应用。

```
    >>> temp = "<html><head><title>%(lang)s<title><body><p>My name
>>> my = {"name":"qiwsir", "lang":"python"}
>>> temp % my
    '<html><head><title>python<title><body><p>My name is qiwsir.</
```

temp就是所谓的模板，双引号所包裹的实质上是一段HTML代码。然后在字典中写好一些数据，按照模板的要求在相应位置显示对应的数据。

是不是一个很有意思的屠龙之技？

1.10.5 相关概念

以下内容是跟字典有关联的基本知识，从维基百科上抄录下来，供读者参考使用。

1.关联数组

在计算机科学中，关联数组（英语：Associative Array）又称映射（Map）、字典（Dictionary）是一个抽象的数据结构，它包含着类似于“键值”的有序对。一个关联数组中的有序对可以重复（如C++中的multimap）也可以不重复（如C++中的map）。

这种数据结构包含以下几种常见的操作：

（1）向关联数组添加配对。

（2）从关联数组内删除配对。

（3）修改关联数组内的配对。

（4）根据已知的键寻找配对。

字典问题是设计一种能够具备关联数组特性的数据结构。解决字典问题的常用方法是利用散列表，但有些情况也可以直接使用有地址的数组、二叉树，或者其他结构。

许多程序设计语言内置基本的数据类型，提供对关联数组的支持。而Content-addressable memory则是硬件层面上实现对关联数组的支持。

2.散列表

散列表（Hash table，也叫哈希表），是根据关键字（Key value）而直接访问在内存存储位置的数据结构。即把键值通过一个函数的计算，映射到表中一个位置来访问记录，加快了查找速度。这个映射函数称作散列函数，存放记录的数组称作散列表。

1.10.6　字典的函数

跟前面所讲述的其他对象类似，字典也有一些函数。通过这些函数，能够实现对字典的操作。

1.copy和deepcopy

拷贝是copy的音译，标准的汉语翻译是“复制”。

在一般的理解中，copy就是将原来的东西再做一份。但是，在Python里面（乃至于很多编程语言中），copy可不是那么简单的。

```
>>> a = 5
>>> b = a
>>> b
5
```

这样做是不是就得到了两个5了呢？表面上看似乎是，但是，不要忘记在前面反复提到的：对象有类型，变量无类型，正是因为这句话，变量其实是一个标签。不妨请出法宝：id()，专门查看内存中的对象编号。

```
>>> id(a)
139774080
>>> id(b)
139774080
```

果然，并没有两个5，就一个，只不过是贴了两张标签而已。这种现象普遍存在于Python的多种数据类型中。其他的就不演示了，就仅看看dict类型。

```
>>> ad = {"name":"qiwsir", "lang":"python"}
>>> bd = ad
>>> bd
{'lang': 'python', 'name': 'qiwsir'}
>>> id(ad)
3072239652L
>>> id(bd)
3072239652L
```

又是一个对象贴了两个标签，这是用赋值的方式实现的所谓“假装拷贝”。那么如果用copy的方法呢？

```
>>> cd = ad.copy()
>>> cd
{'lang': 'python', 'name': 'qiwsir'}
>>> id(cd)
3072239788L
```

这次得到的cd跟原来的ad是不同的，它在内存中另辟了一个空间。如果我尝试修改cd，应该对原来的ad不会造成任何影响。

```
>>> cd["name"] = "itdiffer.com"
>>> cd
{'lang': 'python', 'name': 'itdiffer.com'}
>>> ad
{'lang': 'python', 'name': 'qiwsir'}
```

真的跟推理一模一样。所以，只要理解了“变量”是对象的标签，对象有类型而变量无类型，就能正确推断出Python能够提供的结果。刚才已经看到了，bd和ad引用了同一个内存对象。

```
>>> bd
{'lang': 'python', 'name': 'qiwsir'}
>>> bd["name"] = "laoqi"
>>> ad
{'lang': 'python', 'name': 'laoqi'}
>>> bd
{'lang': 'python', 'name': 'laoqi'}
```

修改了bd所对应的“对象”，ad的“对象”也变了。

然而，事情并没有那么简单，下面看仔细一点，否则就迷茫了。

```
>>> x = {"name":"qiwsir", "lang":["python", "java", "c"]}
>>> y = x.copy()
>>> y
{'lang': ['python', 'java', 'c'], 'name': 'qiwsir'}
>>> id(x)
3072241012L
>>> id(y)
3072241284L
```

y是从x拷贝过来的，两个在内存中是不同的对象。

```
>>> y["lang"].remove("c")
```

在y所对应的字典对象中，键"lang"的值是一个列表，为['python'，'java'，'c']，这里用remove()删除其中的一个元素"c"。删除之后，这个列表变为：['python'，'java']。

```
>>> y
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

那么，x所对应的字典中，这个列表变化了吗？应该没有变化，因为按照前面所讲的，它是另外一个对象，两个互不干扰。

```
>>> x
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

仔细观察，是不是有点出乎意料呢？我没有作弊哦。如果不信，就按照操作自己在交互模式中试试，是不是也能够得到这个结果呢？这是为什么？

但是，如果要操作另外一个键值对：

```
>>> y["name"] = "laoqi"
>>> y
{'lang': ['python', 'java'], 'name': 'laoqi'}
>>> x
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

前面所说的原理是有效的，为什么当值是列表的时候就不奏效了呢？

要破解这个迷局还得用id()：

```
>>> id(x)
3072241012L
>>> id(y)
3072241284L
```

x和y对应着两个不同的对象，的确如此。但这个对象（字典）是由两个键值对组成的，其中一个键的值是列表。

```
>>> id(x["lang"])
3072243276L
>>> id(y["lang"])
3072243276L
```

发现了这样一个事实：列表是同一个对象。

但是，作为字符串为值的那个键值对分属不同的对象。

```
>>> id(x["name"])
3072245184L
>>> id(y["name"])
3072245408L
```

这个事实就说明了为什么修改一个列表，另外一个也跟着修改；而修改一个字符串，另外一个不跟随的原因了。

但是，似乎还没有解开深层的原因。深层的原因与Python存储的对象类型（在不少地方也用“数据类型”的说法，其实两者是一样的，“对象”和“数据”在Python中等同，不用区分）特点有关，Python只存储基本类型的数据，比如int、str，对于不是基础类型的，比如字典的值是列表，Python不会在被复制的那个对象中重新存储，而是用引用的方式，指向原来的值。通俗地说，Python在所执行的复制动作中，如果是基本类型的数据，就在内存中重新建个窝，如果不是基本类型的，就不新建窝了，而是用标签引用原来的窝。即如果比较简单，随便建立新窝即可；但是，如果对象太复杂了，就别费劲了，还是引用一下原来的省事。

所以，把用copy()实现的拷贝称之为“浅拷贝”（不仅Python，很多语言都有“浅拷贝”。顾名思义，没有解决深层次问题。言外之意，还有能够解决深层次问题的方法）。

与“浅拷贝”对应，在Python中，还有一个“深拷贝”（deep copy）。不过，要用import导入一个模块。

```
>>> import copy
>>> z = copy.deepcopy(x)
>>> z
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

用copy.deepcopy()深拷贝了一个新的副本，用id()来勘察一番：

```
>>> id(x["lang"])
3072243276L
>>> id(z["lang"])
3072245068L
```

果然是另外一个“窝”，不是引用了。如果按照这个结果，修改其中一个列表中的元素，应该就不影响另外一个了。

```
>>> x
{'lang': ['python', 'java'], 'name': 'qiwsir'}
>>> x["lang"].remove("java")
>>> x
{'lang': ['python'], 'name': 'qiwsir'}
>>> z
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

果然如此，再试试才过瘾呀。

```
>>> x["lang"].append("c++")
>>> x
{'lang': ['python', 'c++'], 'name': 'qiwsir'}
```

这就是所谓的浅拷贝和深拷贝。

2.clear

在交互模式中，用help是一个很好的习惯。

```
>>> help(dict.clear)
clear(...)
    D.clear() -> None.  Remove all items from D.
```

这是一个清空字典中所有元素的操作。

```
>>> a = {"name":"qiwsir"}
>>> a.clear()
>>> a
{}
```

clear的含义是将字典清空，得到的是“空”字典。它和del有着很大的区别，del是将字典删除，内存中就没有它了，并不是为“空”（“空”和无是有区别的，至少在编程语言中有区别。在其他方面也有区别，比如“色即是空”，不能是“无”吧）。

```
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

果然删除了。

另外，如果要清空一个字典，还能够使用a＝{}这种方法，但这种方法的本质是将变量a转向了{}这个对象，那么原来的呢？原来的成为了断线的风筝。这样的东西在Python中称之为垃圾，而且Python能够自动将这样的垃圾回收。读者就不用关心它了，反正Python会处理的。

3.get和setdefault

get的含义是：

```
get(...)
    D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
```

注意，在这个说明中，“if k in D”就返回其值。

```
>>> d
{'lang': 'python'}
>>> d.get("lang")
'python'
```

dict.get()就是要得到字典中某个键的值，只是它没有那么“严

历”罢了。因为类似获得字典中键值的方法，如d['lang']就能得到对应的值"python"，可是，如果要获取的键不存在：

```
>>> print d.get("name")
None

>>> d["name"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'name'
```

这就是dict.get()和dict['key']的区别。

如果键不在字典中，会返回None，这是一种情况，另外还可以这样：

```
>>> d = {"lang":"python"}
>>> newd = d.get("name",'qiwsir')
>>> newd
'qiwsir'
>>> d
{'lang': 'python'}
```

以d.get（"name"，'qiwsir'）的方式，如果不能得到键"name"的值，就返回后面指定的值"qiwsir"。这就是文档中D[k]if k in D，else d.的含义，这样做并没有影响原来的字典。

另外一个跟get在功能上有相似地方的是D.setdefault（k），其含义是：

```
setdefault(...)
    D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in
```

首先，它要执行D.get（k，d）就跟前面一样了，然后，进一步执行另外一个操作，如果键k不在字典中，就在字典中增加这个键值对。当然，如果有就没有必要执行这一步了。

```
>>> d
{'lang': 'python'}
>>> d.setdefault("lang")
'python'
```

在字典中，有"lang"这个键，就返回它的值。

```
>>> d.setdefault("name","qiwsir")
'qiwsir'
>>> d
{'lang': 'python', 'name': 'qiwsir'}
```

在字典中没有"name"这个键，于是返回d.setdefault（"name"，"qiwsir"）指定的值"qiwsir"，并且将键值对'name':"qiwsir"添加到原来的字典中。

如果这样操作：

```
>>> d.setdefault("web")
```

什么也没有返回吗？不是，返回了，只不过没有显示出来，如果你用print就能看到了。因为这里返回的是一个None，不妨查看一下那个字典：

```
>>> d
{'lang': 'python', 'web': None, 'name': 'qiwsir'}
```

键"web"的值成为了None。

4.items/iteritems，keys/iterkeys，values/itervalues

这个标题中列出的是三组函数，并且这三组有相似的地方。在这里详细讲述一下第一组，其余两组，我想凭借读者的聪明智慧是不在话下的。

```
>>> help(dict.items)
items(...)
    D.items() -> list of D's (key, value) pairs, as 2-tuples
```

这种获取帮助信息的方法是惯用的伎俩了，希望读者能熟悉，并在自己的代码生涯中常用。D.items()能够得到一个关于字典的列表，列表中的元素是由字典中的键和值组成的元组。例如：

```
>>> dd = {"name":"qiwsir", "lang":"python", "web":"www.itdiffer.co
>>> dd_kv = dd.items()
>>> dd_kv
[('lang', 'python'), ('web', 'www.itdiffer.com'), ('name', 'qiwsir
```

显然，是有返回值的。这个操作在后面要讲到的循环中将有很大的作用。

跟items类似的是iteritems，这个词的特点是由iter和items拼接而成的，后部分items就不用说了，肯定是在告诉我们，得到的结果跟D.items()的结果类似。还有一个iter是什么意思？前面我提到了一个词“iterable”，它的含义是“可迭代的”，这里的iter是指名词iterator的前部分，意思是“迭代器”。合起来，"iteritems"的含义就是：

```
iteritems(...)
    D.iteritems() -> an iterator over the (key, value) items of D
```

你看，学习Python不是什么难事，只要充分使用帮助文档就好了。这里告诉我们，得到的是一个“迭代器”（关于什么是迭代器，以及相关的内容，后续会详细讲述），这个迭代器是关于“D.items()”的。看个例子就明白了。

```
>>> dd
{'lang': 'python', 'web': 'www.itdiffer.com', 'name': 'qiwsir'}
>>> dd_iter = dd.iteritems()
>>> type(dd_iter)
<type 'dictionary-itemiterator'>
>>> dd_iter
<dictionary-itemiterator object at 0xb72b9a2c>
>>> list(dd_iter)
[('lang', 'python'), ('web', 'www.itdiffer.com'), ('name', 'qiwsir
```

得到的dd_iter是一个'dictionary-itemiterator'类型，不过这种迭代器类型的数据不能直接输出，必须用list()转换一下，才能看到里面的真面目。

另外两组含义跟这个相似，只不过是得到key或者value。下面仅列举一下例子，具体内容，读者可以自行在交互模式中看文档。

```
>>> dd
{'lang': 'python', 'web': 'www.itdiffer.com', 'name': 'qiwsir'}
>>> dd.keys()
['lang', 'web', 'name']
>>> dd.values()
['python', 'www.itdiffer.com', 'qiwsir']
```

这里先交代一句，如果要实现对“键值”对或者“键”或者“值”的循环，用迭代器的效率会高一些。对这句话的理解，继续阅读本书就能

找到解释。

5.pop和popitem

还记得删除列表中元素的函数是哪个吗？pop和remove，这两个的区别在于：list.remove（x）用来删除指定的元素，而list.pop（[i]）用于删除指定索引的元素，如果不提供索引值，就默认删除最后一个。

在字典中，也有删除键值对的函数。

```
pop(...)
    D.pop(k[,d]) -> v, remove specified key and return the corresp
    If key is not found, d is returned if given, otherwise KeyErro
```

D.pop（k[，d]）是以字典的键为参数，删除指定键的键值对，当然，如果输入对应的值也可以，那个是可选的。

```
>>> dd
{'lang': 'python', 'web': 'www.itdiffer.com', 'name': 'qiwsir'}
>>> dd.pop("name")
'qiwsir'
```

删除指定键"name"，返回了其值"qiwsir"。这样，在原字典中，“'name':'qiwsir'”这个键值对就被删除了。

```
>>> dd
{'lang': 'python', 'web': 'www.itdiffer.com'}
```

值得注意的是，pop函数中的参数是不能省略的，这跟列表中的pop有所不同。

```
>>> dd.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pop expected at least 1 arguments, got 0
```

如果要删除字典中没有的键值对，也会报错。

```
>>> dd.pop("name")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
KeyError: 'name'
```

有意思的是D.popitem()跟list.pop()有相似之处，不用写参数（list.pop()可以不写参数），但是，D.popitem()不是删除最后一个，dict没有顺序，也就没有最后和最先了，它是随机删除一个，并将所删除的返回。

```
popitem(...)
    D.popitem() -> (k, v), remove and return some (key, value) pai
    2-tuple; but raise KeyError if D is empty.
```

如果字典是空的，就要报错了

```
>>> dd
{'lang': 'python', 'web': 'www.itdiffer.com'}
>>> dd.popitem()
('lang', 'python')
>>> dd
{'web': 'www.itdiffer.com'}
```

成功地删除了一对，注意是随机的，不是删除前面显示的最后一个，你做同样的操作，或许删除的对象跟我删除的不一样。并且返回了删除的内容，返回值是元组类型，且其元素为所删除的键和值。

```
>>> dd.popitems()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'popitems'
```

错了？注意看提示信息，果然错了。注意是popitem，不要多了s，前面的D.items()中包含s，是复数形式，说明它能够返回多个结果（多个元组组成的列表），而在D.popitem()中，一次只能随机删除一对键值对，并以一个元组的形式返回，所以，要用单数形式，不能用复数形式了。

```
>>> dd.popitem()
('web', 'www.itdiffer.com')
>>> dd
{}
```

都删了，字典成空的了。如果再删，会怎么样？

```
>>> dd.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

报错信息中明确告知，字典已经是空的了，没有能删的东西了。

6.update

update()，看名字就猜测到一二了，是不是更新字典内容呢？的确是。

```
update(...)
    D.update([E, ]**F) -> None.  Update D from dict/iterable E and
    If E present and has a .keys() method, does:     for k in E: D
    If E present and lacks .keys() method, does:     for (k, v) in
    In either case, this is followed by: for k in F: D[k] = F[k]
```

看样子这个函数有点复杂，不要着急，通过实验可以一点一点鼓捣明白。

首先，这个函数返回值是None，它的作用就是更新字典。其参数可以是字典或者某种可迭代的对象。

```
>>> d1 = {"lang":"python"}
>>> d2 = {"song":"I dreamed a dream"}
>>> d1.update(d2)
>>> d1
{'lang': 'python', 'song': 'I dreamed a dream'}
>>> d2
{'song': 'I dreamed a dream'}
```

这样就把字典d2更新纳入了d1那个字典，于是d1中就多了一些内容，因为把d2的内容包含进来了。当然d2还存在，并没有受到影响。

还可以用下面的方法更新：

```
>>> d2
{'song': 'I dreamed a dream'}
>>> d2.update([("name","qiwsir"), ("web","itdiffer.com")])
>>> d2
{'web': 'itdiffer.com', 'name': 'qiwsir', 'song': 'I dreamed a dre
```

列表内的元组是键值对。

7.has_key

这个函数的功能是判断字典中是否存在某个键。

```
has_key(...)
    D.has_key(k) -> True if D has a key k, else False
```

跟前一节中遇到的k in D类似。

```
>>> d2
{'web': 'itdiffer.com', 'name': 'qiwsir', 'song': 'I dreamed a dre
>>> d2.has_key("web")
True
>>> "web" in d2
```

关于dict的函数似乎不少。不用着急，也不用担心记不住，因为根本不需要记忆，只要会用搜索即可。

1.11 集合

Python是一个发展的语言，尽管前面已经有好几种对象类型了，但是，从实践的角度看还不够，所以，又有了名曰“集合”的这种类型。当然，对象类型本质上是自己可以定义的，要想学会自己定义，请继续阅读本书，不要半途而废，“而世之奇伟、瑰怪，非常之观，常在于险远，而人之所罕至焉，故非有志者不能至也。”（王安石《游褒禅山记》）。

另外，也不要担心记不住，你只要记住爱因斯坦说的就好了：

爱因斯坦在美国演讲，有人问：“你可记得声音的速度是多少？你如何记下许多东西？”

爱因斯坦轻松答道：“声音的速度是多少，我必须查辞典才能回答。因为我从来不记在辞典上已经印着的东西，我的记忆力是用来记忆书本上没有的东西。”

多么霸气的回答，这回答不仅霸气，还告诉我们一种方法：只要是能够通过某种方法查找到的，就不需要记忆。

各种对象类型都可以通过下述方法但不限于这些方法查到：

- 交互模式下用dir()或者help()。
- 使用Google。

上述方法从本书开始就不断强调和示范，目的就在于提示读者要掌握方法，而不是记忆知识。

对学过的对象类型做个归纳整理：

- 能够索引的，如list/str，其中的元素可以重复。
- 可变的，如list/dict，即其中的元素/键值对可以原地修改。
- 不可变的，如str/int，即不能进行原地修改。
- 无索引序列的，如dict，即其中的元素（键值对）没有排列顺序。

1.11.1 创建集合

集合的英文是set，翻译过来叫作“集合”。它的特点是：有的可变，有的不可变；元素无次序，不可重复。

如果说元组（tuple）算是列表（list）和字符串（str）的杂合，那么集合（set）则可以堪称是list和dict的杂合。

集合拥有类似字典的特点：可以用{}花括号来定义；其中的元素没有序列，也就是非序列类型的数据；而且集合中的元素不可重复，这就类似于dict键。

集合也有一点列表的特点：有一种集合可以在原处修改。

通过实验，逐步理解创建set的方法：

```
>>> s1 = set("qiwsir")
>>> s1
set(['q', 'i', 's', 'r', 'w'])
```

把字符串中的字符拆解开形成了集合。特别注意观察：qiwsir中有两个i，但是在s1中只有一个i，也就是集合中元素不能重复。

```
>>> s2 = set([123, "google", "face", "book", "facebook", "book"])
>>> s2
set(['facebook', 123, 'google', 'book', 'face'])
```

在创建集合的时候，如果发现了重复的元素，就会过滤一下，剩下不重复的。而且，从s2的创建可以看出，查看结果时显示的元素排列顺序与开始建立时不同，完全是随意显示的（怎么能说明是随机的呢？读者有没有办法？），这说明集合中的元素没有序列。

```
>>> s3 = {"facebook", 123}        #通过{}直接创建
>>> s3
set([123, 'facebook'])
```

除了用set()来创建集合，还可以使用{}的方式，但是这种方式不提倡使用，因为在某些情况下，Python搞不清楚是字典还是集合。看看下面的探讨就发现问题了。

```
>>> s3 = {"facebook", [1,2,'a'], {"name":"python", "lang":"english
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'

>>> s3 = {"facebook", [1,2], 123}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

从上述实验可以看出，通过{}无法创建含有列表或者字典类型对象元素的集合。

认真阅读报错信息，有这样的词汇："unhashable"，在理解这个词之前，先看它的反义词"hashable"，很多时候翻译为"可哈希"，其实它有一个不是音译的名词"散列"。如果我们简单点理解，某数据"不可哈希"（unhashable）就是其可变，如列表和字典都能原地修改，就是unhashable。否则，不可变的，类似字符串那样不能原地修改的就是hashable（可哈希）。

对于字典类型的对象，其"键"必须是hashable，即不可变。

现在遇到的集合，其元素也是"可哈希"的。上面的例子，试图将字典、列表作为元素的元素，就报错了。而且报错信息中明确告知列表和字典是不可哈希类型，言外之意，里面的元素都应该是可哈希类型。

继续探索另外一种情况：

```
>>> s1
set(['q', 'i', 's', 'r', 'w'])
>>> s1[1] = "I"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item assignment
```

这里报错，进一步说明集合不是序列类型，不能用索引方式对其进行修改。

根据前面的经验，类型名称函数能够实现类型转换，比如str()就是将对象转化为字符串，同理，分别用list()和set()能够实现集合和列表两种对象之间的转化。

```
>>> s1
```

```
set(['q', 'i', 's', 'r', 'w'])
>>> lst = list(s1)
>>> lst
['q', 'i', 's', 'r', 'w']
>>> lst[1] = "I"
>>> lst
['q', 'I', 's', 'r', 'w']
```

特别说明，利用set()建立起来的集合是可变集合，可变集合都是unhashable类型的。

1.11.2 集合的函数

把与集合有关的函数找出来。

```
>>> dir(set)
    ['__and__', '__class__', '__cmp__', '__contains__', '__delattr
```

为了看清楚，我把双画线“__”先删除掉：

```
'add', 'clear', 'copy', 'difference', 'difference_update', 'discar
```

然后用help()可以找到每个函数的具体使用方法。

1.add和update

```
>>> help(set.add)

Help on method_descriptor:

add(...)
Add an element to a set.
This has no effect if the element is already present.
```

在交互模式中，可以看到：

```
>>> a_set = {}              #我想当然地认为这样也可以建立一个set
>>> a_set.add("qiwsir")     #报错！看错误信息。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'add'
```

```
>>> type(a_set)              #Python认为我建立的是一个dict
<type 'dict'>
```

{}这个东西在dict和set中都有用，但是，若按照上面的方法则建立的是dict，而不是set。这是Python规定的，要建立set，只能用前面已经看到的创建方法了。

```
>>> a_set = {'a','i'}
>>> type(a_set)
  <type 'set'>

>>> a_set.add("qiwsir")      #增加一个元素
>>> a_set                    #原地修改
set(['i', 'a', 'qiwsir'])

>>> b_set = set("python")
>>> type(b_set)
<type 'set'>
>>> b_set
set(['h', 'o', 'n', 'p', 't', 'y'])
>>> b_set.add("qiwsir")
>>> b_set
set(['h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])

>>> b_set.add([1,2,3])       #列表是不可哈希的，集合中的元素应该是hashabl
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

>>> b_set.add('[1,2,3]')     #可以这样!
>>> b_set
set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
```

除了add()之外，还能够从另外一个集合中合并过来元素，方法是set.update（s2）。

```
>>> help(set.update)
update(...)
    Update a set with the union of itself and others.

>>> s1
set(['a', 'b'])
>>> s2
set(['github', 'qiwsir'])
>>> s1.update(s2)
>>> s1
set(['a', 'qiwsir', 'b', 'github'])
```

```
>>> s2
set(['github', 'qiwsir'])
```

2. pop, remove, discard, clear

```
>>> help(set.pop)
pop(...)
    Remove and return an arbitrary set element.
    Raises KeyError if the set is empty.

>>> b_set
set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
>>> b_set.pop()          #从set中任意选一个删除，并返回该值
'[1,2,3]'
>>> b_set.pop()
'h'
>>> b_set.pop()
'o'
>>> b_set
set(['n', 'p', 't', 'qiwsir', 'y'])

>>> b_set.pop("n")       #如果要指定删除某个元素就报错了
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pop() takes no arguments (1 given)
```

set.pop()是从集合中随机选一个元素删除并将这个值返回，但是不能指定删除某个元素。报错信息告诉我们，pop()不能有参数。此外，如果集合是空的了，再做pop()操作也报错。

但是否可以删除指定元素？如果可以，怎么办？

```
>>> help(set.remove)

remove(...)
    Remove an element from a set; it must be a member.
    If the element is not a member, raise a KeyError.
```

会有办法的，那么多聪明人早就为读者想好了一个函数——remove()——set.remove（obj）中的obj，必须是set中的元素，否则就报错。

```
>>> a_set
set(['i', 'a', 'qiwsir'])
```

```
>>> a_set.remove("i")
>>> a_set
set(['a', 'qiwsir'])
>>> a_set.remove("w")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'w'
```

跟remove（obj）类似的还有discard（obj）：

```
>>> help(set.discard)

discard(...)
    Remove an element from a set if it is a member.
    If the element is not a member, do nothing.
```

与help（set.remove）进行信息对比，看看有什么不同。关键在于如果discard（obj）中的obj是set中的元素就删除，如果不是，就什么也不做。新闻就要对比着看才有意思，这里也一样。

```
>>> a_set.discard('a')
>>> a_set
set(['qiwsir'])
>>> a_set.discard('b')
>>>
```

在删除上还有一个绝杀，就是set.clear()，它的功能是：Remove all elements from this set.（自己在交互模式下help（set.clear））。

```
>>> a_set
set(['qiwsir'])
>>> a_set.clear()
>>> a_set
set([])
>>> bool(a_set)      #空了,bool一下返回False.
False
```

1.11.3 补充知识

集合也是一个数学概念（以下定义来自维基百科）：

最简单的说法是最原始的集合论——朴素集合论中的定义，集合

就是“一堆东西”。集合里的“东西”叫作元素。若然x是集合A的元素，记作x∈A。

集合是现代数学中一个重要的基本概念。集合论的基本理论直到19世纪末才被创立，现在已经是数学教育中一个普遍存在的部分，在小学时就开始学习了。这里对被数学家们称为“直观的”或“朴素的”集合论进行一个简短而基本的介绍；更详细的分析可见朴素集合论。对集合进行严格的公理推导可见公理化集合论。

在计算机中集合是什么呢？自维基百科是这么说的：

在计算机科学中，集合是一组可变数量的数据项（也可能是0个）的组合，这些数据项可能共享某些特征，需要以某种操作方式一起进行操作。一般来讲，这些数据项的类型是相同的，或基类相同（若使用的语言支持继承）。列表（或数组）通常不被认为是集合，因为其大小固定，但事实上它常常在实现中作为某些形式的集合使用。

集合的种类包括列表、集、多重集、树和图。枚举类型可以是列表或集。

1.11.4 不变的集合

以set()创立的集合都是可原地修改的集合，或者说是可变的，也可以说是unhashable。

还有一种集合不能原地修改，这种集合的创建方法是用frozenset()，顾名思义，这是一个被“冻结”的集合，当然是不能修改的，这种集合就是hashable类型——可哈希。

```
>>> f_set = frozenset("qiwsir")
>>> f_set
frozenset(['q', 'i', 's', 'r', 'w'])
>>> f_set.add("python")                      #报错，不能修改
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'

>>> a_set = set("github")                    #对比看一看
>>> a_set
set(['b', 'g', 'i', 'h', 'u', 't'])
```

```
>>> a_set.add("python")
>>> a_set
set(['b', 'g', 'i', 'h', 'python', 'u', 't'])
```

1.11.5 集合运算

唤醒中学数学（准确说是高中数学中的一点知识）中关于集合的知识，当然，如果你是某个理工科的专业大学毕业，应该更熟悉集合之间的关系。

1.元素与集合的关系

元素与集合就一种关系，要么属于某个集合，要么不属于。

```
>>> aset
set(['h', 'o', 'n', 'p', 't', 'y'])
>>> "a" in aset
False
>>> "h" in aset
```

2.集合与集合的关系

假设两个集合A、B

（1）A是否等于B，即两个集合的元素是否完全一样。

在交互模式下实验

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a == b
False
>>> a != b
```

（2）A是否是B的子集，或者反过来，B是否是A的超集，即A的元素是否也都是B的元素，且B的元素比A的元素数量多。

判断集合A是否是集合B的子集，可以使用A<B，返回True则是

子集，否则不是。另外，还可以使用函数A.issubset（B）判断。

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> c
set(['q', 'i'])
>>> c < a              #c是a的子集
True
>>> c.issubset(a)         #或者用这种方法，判断c是否是a的子集
True
>>> a.issuperset(c)       #判断a是否是c的超集
True

>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a < b              #a不是b的子集
False
>>> a.issubset(b)
False
```

（3）A、B的并集，即A、B所有元素，如图1-8所示。

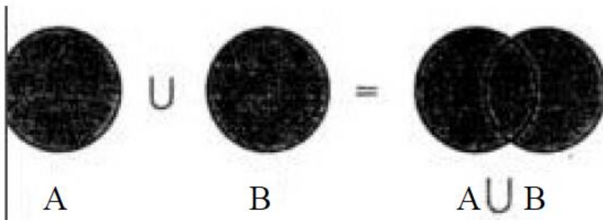


图1-8　A、B的并集

可以使用的符号是“|”，是一个半角状态下的竖线，输入方法是在英文状态下，按下“shift”加上右方括号右边的那个键。表达式是A|B也可使用函数A.union（B），得到的结果就是两个集合并集，注意，这个结果是新生成的一个对象，不是将结合A扩充。

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a | b                      #可以有两种方式，结果一样
set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])
>>> a.union(b)
set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])
```

（4）A、B的交集，即A、B所公有的元素，如图1-9所示。

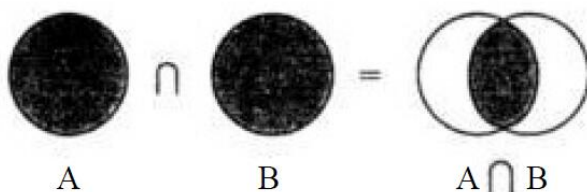


图1-9　A、B的交集

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a & b                    #两种方式，等价
set(['q', 'i'])
>>> a.intersection(b)
set(['q', 'i'])
```

我在实验的时候，顺手敲了下面的代码，出现的结果如下，读者能解释一下吗？

```
>>> a and b
set(['a', 'q', 'i', 'l', 'o'])
```

（5）A相对B的差（补），即A相对B不同的部分元素，如图1-10所示。

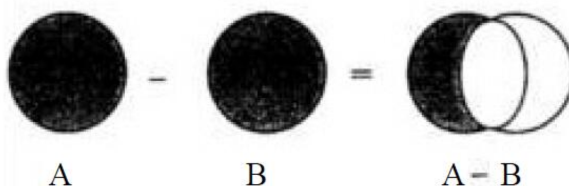


图1-10　A相对B的差（补）

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a - b
```

```
set(['s', 'r', 'w'])
>>> a.difference(b)
set(['s', 'r', 'w'])
```

（6）A、B的对称差集，如图1-11所示。

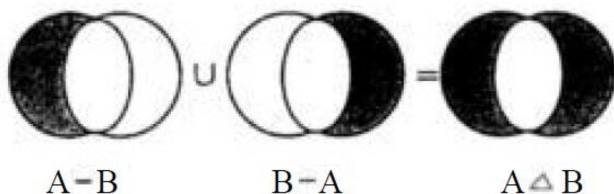


图1-11　A、B的对称差集

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a.symmetric_difference(b)
set(['a', 'l', 'o', 's', 'r', 'w'])
```

以上是集合的基本运算。在编程中如果用到，可以用前面说的方法查找。

第2章　语句和文件

“呀呀学语”是小孩子成长的必经阶段，学习编程，到这里就是“呀呀学语”的阶段。从本章开始，将使用已经学习过的基本知识，组装成各种语句，通过语句向计算机传达自己的意愿。所以，读者将从本章收获到“呀呀学语”的回忆。

2.1 运算符

在编程语言中运算符是比较多样化的，虽然已经在前面了解了各种运算符和其优先级，但是，那时对Python的理解还比较肤浅，现在要温故而知新，更上一层楼。

2.1.1 算术运算符

四则运算中的一些运算符，如加减乘除，对应的符号分别是：＋、-、*、／，此外，还有求余数的“%”等，都是算术运算符。

列出一个表格，将所有的运算符表现出来。不用记，但是要认真地看一看，知道有哪些，如果以后用到，可以来查，如表2-1所示。

表2-1 算术运算符

运算符	描　　述	实　　例
+	加，两个对象相加	10+20 输出结果 30
–	减，得到负数或是一个数减去另一个数	10–20 输出结果 –10
*	乘，两个数相乘或是返回一个被重复若干次的字符串	10 * 20 输出结果 200
/	除，x 除以 y	20/10 输出结果 2
%	取余，返回除法的余数	20%10 输出结果 0
**	幂，返回 x 的 y 次幂	10**2 输出结果 100
//	取整除，返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

读者可以根据中学的数学知识，想想上面的运算符在混合运算中应该按照什么顺序计算，并且亲自试试，是否与中学数学中的规律一致。

2.1.2 比较运算符

所谓比较，就是将两个东西相比，比如做家长的经常把自己的孩子跟别人的孩子比较，唯恐自己的孩子在某方面差了。

在计算机高级语言编程中，任何两个同一类型的量都可以进行比较，比如两个数字可以比较，两个字符串可以比较。不同类型的量可以比较吗？这种比较没有意义，比如，二两肉和三尺布如何进行比

较？所以，在真正的编程中，我们要谨慎对待这种不同类型量的比较。

但是，在某些语言中，允许这种比较，因为它们在比较的时候，都是将非数值类型转化为数值类型比较。

对于比较运算符，在小学数学中就学习了一些：大于、小于、等于、不等于。没有陌生的东西，Python里面也是如此，如表2-2所示。

以下假设变量a为10，变量b为20。

表2-2　比较运算符

运算符	描　述	实　例
==	等于（注意：两个符号）	(a == b) 返回 False
!=	不等于	(a != b) 返回 True
>	大于	(a > b) 返回 False
<	小于	(a < b) 返回 True
>=	大于等于	(a >= b) 返回 False
<=	小于等于	(a <= b) 返回 True

在表2-2中，显示比较的结果就是返回True或者False，即这个比较如果成立，就为真，返回True，若返回False，则说明比较不成立。

请按照下面的方式进行比较操作，然后再根据自己的想象进行练习。

```
>>> a=10
>>> b=20
>>> a > b
False
>>> a < b
True
>>> a == b
False
>>> a != b
True
>>> a >= b
False
>>> a <= b
True
```

除了数字之外，还可以对字符串进行比较。字符串中的比较是按照“字典顺序”进行比较的，当然，这里说的是英文的字典。

```
>>> a = "qiwsir"
>>> b = "python"
>>> a > b
True
```

先看第一个字符，按照字典顺序，q大于p（在字典中，q排在p的后面），那么就返回结果True。

在Python中，某些两种不同类型的对象，虽然可以进行比较，但是不赞成这样进行比较。

```
>>> a = 5
>>> b = "5"
>>> a > b
False
```

2.1.3 逻辑运算符

首先谈谈什么是逻辑，韩寒先生对逻辑有一个分类：

逻辑分两种，一种是逻辑，另一种是中国人的逻辑。

这种分类的确非常精准。在很多情况下，中国人有很奇特的逻辑。但是，在Python中，讲的是逻辑，不是中国人的逻辑。

逻辑（logic），又称理则、论理、推理、推论，是有效推论的哲学研究。逻辑被使用在大部分智能活动中，但主要在哲学、数学、语义学和计算机科学等领域内被视为一门学科。在数学里，逻辑是指研究某个形式语言的有效推论。

1.布尔类型的变量

在所有的高级编程语言中都有一类对象类型，被称之为布尔型，这是用一个人的名字来命名的。

乔治·布尔（George Boole，1815年 - 1864年），英格兰数学家、哲学家。

乔治·布尔是一个皮匠的儿子，生于英格兰的林肯。由于家境贫寒，布尔不得不在协助养家的同时为自己能受教育而奋斗，不管怎么

说，他成了19世纪最重要的数学家之一。尽管他考虑过以牧师为业，但最终还是决定从教，而且不久就开办了自己的学校。

在备课的时候，布尔不满意当时的数学课本，便决定阅读伟大数学家的论文。在阅读伟大的法国数学家拉格朗日的论文时，布尔有了变分法方面的新发现。变分法是数学分析的分支，它处理的是寻求优化某些参数的曲线和曲面。

1848年，布尔出版了《The Mathematical Analysis of Logic》，这是他对符号逻辑诸多贡献中的第一次。

1849年，他被任命于爱尔兰科克的皇后学院（今科克大学或UCC）的数学教授。1854年，他出版了《The Laws of Thought》，这是他最著名的著作。在这本书中布尔介绍了以他的名字命名的布尔代数。布尔撰写了微分方程和差分方程的课本，这些课本在英国一直使用到19世纪末。

由于其在符号逻辑运算中的特殊贡献，很多计算机语言中将逻辑运算称为布尔运算，将其结果称为布尔值。

布尔所创立的这套逻辑被称之为“布尔代数”，其中规定只有两种值，True和False，正好对应计算机上二进制数的1和0。所以，布尔代数和计算机是天然吻合的。

所谓布尔类型就是返回结果为1（True）或0（False）的数据变量。

在Python中（其他高级语言也类似）有三种运算符，可以实现布尔类型的对象间的运算。

2.布尔运算

布尔运算符如表2-3所示。

表2-3　布尔运算符

运算符	描　述	实　例
and	“与”	如果 x 为 False，x and y，返回 False，否则返回 y 的计算值
or	“或”	如果 x 是 True，x or y，返回 True，否则返回 y 的计算值
not	布尔“非”	如果 x 为 True，not x，返回 False。如果 x 为 False，返回 True

（1）and

and，翻译为“与”，但事实上，这种翻译容易引起望文生义的理解。先说一下正确的理解。A and B，含义是：首先运算A，如果A的值是True，就计算B，并将B的结果返回做为最终结果（如果B是False，那么A and B的最终结果就是False，如果B的结果是True，那么A and B的结果就是True）；如果A的值是False，就不计算B了，直接返回False作为A and B的结果。

比如：

4＞3 and 4＜9，首先看4＞3的值，这个值是True，再看4＜9的值，是True，那么最终这个表达式的结果为True。

```
>>> 4 > 3 and 4 < 9
True
```

4＞3 and 4＜2，先看4＞3，返回True，再看4＜2，返回的是False，那么最终结果是False。

```
>>> 4 > 3 and 4 < 2
False
```

4＜3 and 4＜9，先看4＜3，返回为False，就不看后面的了，直接返回这个结果作为最终结果（对这种现象，有一个形象的说法，叫作“短路”）。

```
>>> 4 < 3 and 4 < 2
False
```

前面说容易引起望文生义的理解，就是有很多人认为无论什么时候都看and两边的值，若值都是True则返回True，若有一个是False就返回False。根据这种理解得到的结果与前述理解得到的结果一样，但是，运算量不一样。

（2）or

or，翻译为“或”。在A or B中，它是这么运算的：

```
if A == True:
    return True
else:
    return B
```

上面这段算是伪代码，所谓伪代码，就不是真正的代码，无法运行。但是，伪代码也有用途，就是能够以类似代码的方式表达一种计算过程。本书读者应该能对这段伪代码有一个大概的理解。

下面再加上每行的注释。这个伪代码跟自然的英语差不多了。

```
if A==True:             #如果A的值是True
    return True         #返回True，表达式最终结果是True
else:                   #否则，也就是A的值不是True
    return B            #看B的值，然后就返回B的值作为最终结果
```

根据上面的运算过程举例。

```
>>> 4 < 3 or 4 < 9
True
>>> 4 < 3 or 4 > 9
False
>>> 4 > 3 or 4 > 9
True
```

（3）not

not，翻译成“非”，窃以为非常好，不论面对什么，就是要否定它。

```
>>> not(4>3)
False
>>> not(4<3)
True
```

关于运算符问题，其实不止上面这些，比如还有成员运算符in，在后面的学习中会逐渐遇到。

2.2 简单语句

学习编程序，就好比小学生学习写作一样，先学会一些词语，就等同于已经掌握了基本的对象类型，接下来就是学“造句子”的时候了。

在编程语言中，句子被称之为“语句”。

事实上，前面已经用过语句了，最典型的print"Hello，World"就是语句。

为了能够严谨地阐述这个概念，抄一段维基百科中的词条：命令式编程。

命令式编程（英语：Imperative programming），是一种描述电脑所需做出的行为的编程范型。几乎所有电脑的硬件工作都是指令式的；几乎所有电脑的硬件都是设计来运行机器码，使用指令式的风格来写的。较高级的指令式编程语言使用变量和更复杂的语句，但仍依从相同的范型。

运算语句一般来说都表现了在存储器内的数据进行运算的行为，然后将结果存入存储器中以便日后使用。高级命令式编程语言更能处理复杂的表达式，可能会产生四则运算和函数计算的结合。

一般高级语言都包含如下语句，Python也不例外。

（1）循环语句：容许一些语句反复运行数次。可依据一个默认的数目来决定运行这些语句的执行次数；或反复运行它们，直至某些条件改变。

（2）条件语句：容许仅当某些条件成立时才运行某个区块。否则，这个区块中的语句会略去，然后按区块后的语句继续运行。

（3）无条件分支语句：容许运行顺序转移到程序的其他部分之中。包括跳跃（在很多语言中称为Goto）、副程序和Procedure等。

循环、条件分支和无条件分支都是控制流程。

当然，Python中的语句还是有自己的特别之处的（别的语言也有自己的特色）。下面就开始娓娓道来。

2.2.1 print

在Python 2.x中，print是一个语句，但是在Python 3.x中它就是一个函数了。不过，这里所演示的还是Python 2.x。

print发起的语句，在程序中主要是将某些东西打印出来，还记得在字符串的格式化输出吗？那就是用来print的。

```
>>> print "hello, world"
hello, world
>>> print "hello", "world"
hello world
```

请仔细观察上面两个print语句的差别。第一个打印的是“hello，world”，包括其中的逗号和空格，是一个完整的字符串。第二个打印的是两个字符串，一个是“hello”，另外一个是“world”，两个字符串之间用逗号分隔。

本来，在print语句中，字符串后面会接一个\n符号，即换行。但是，如果要在一个字符串后面跟着逗号，那么换行就取消了，意味着两个字符串“hello”和“world”打印在同一行。

或许现在体现得还不是很明显，换一个方法就显示出来了。（下面用到了一个称为for循环的语句，后面要详细讲述。）

```
>>> for i in [1,2,3,4,5]:
...     print i
...
1
2
3
4
5
```

这个循环的意思就是要从列表中依次取出每个元素，然后赋值给变量i，并用print语句打印打出来。在变量i后面没有任何符号，每打印一个就换行，再打印下一个，这就是那个“\n”起的作用。

下面的方式就跟上面的有点区别了。

```
>>> for i in [1,2,3,4,5]:
...     print i ,
...
1 2 3 4 5
```

在print语句的最后加了一个逗号，打印出来的就在一行了。

print语句经常用在调试程序的过程，让我们能够知道程序在执行过程中产生的结果。

2.2.2 import

曾经用到过一个Python标准库math，它能提供很多数学函数，但是这些函数不是Python的内建函数，而是属于math的，所以，要用import math来引入这个标准库。

这种用import引入模块（或者库、包）的方法是Python编程经常用到的。引用方法有如下几种：

```
>>> import math
>>> math.pow(3,2)
9.0
```

这是常用的一种方式，而且非常明确，math.pow（3，2）就明确显示了，pow()函数是math模块里的。可以说这是一种可读性非常好的引用方式，并且不同模块的同名函数不会产生冲突。

```
>>> from math import pow
>>> pow(3,2)
9.0
```

这种方法就有点偷懒了，不过也不难理解，从字面意思就知道pow()函数来自于math模块。在后续使用的时候，只需要直接使用pow()即可，不需要在前面写上模块名称了。这种引用方法，比较适合于引入模块较少的时候。如果引入模块多了，可读性就下降了，会不知道哪个函数来自哪个模块。

```
>>> from math import pow as pingfang
```

```
>>> pingfang(3,2)
9.0
```

这是在前面基础上的发展，把从某个模块引入的函数重新命名，比如讲pow重命名为pingfang，然后使用pingfang()就相当于使用pow()了。

还会遇到要引入多个函数的情况，可以这样做：

```
>>> from math import pow, e, pi
>>> pow(e,pi)
23.140692632779263
```

引入了math模块里面的pow、e、pi，pow()是一个乘方函数，e就是欧拉数；pi就是π。

e，作为数学常数，是自然对数函数的底数。有时称它为欧拉数（Euler's number），以瑞士数学家欧拉命名；还有个较鲜见的名字是纳皮尔常数，以纪念苏格兰数学家约翰·纳皮尔引进对数。它是一个无限不循环小数。e＝2.71828182845904523536（《维基百科》）

e的π次方，是一个数学常数，与e和π一样是一个超越数。这个常数在希尔伯特第七问题中曾提到过。（《维基百科》）

```
>>> from math import *
>>> pow(3,2)
9.0
>>> sqrt(9)
3.0
```

这种引入方式是最省事的了，一下将math中的所有函数都引过来了。“好事成双”往往都是梦想，这样引入模块中的函数，其可读性难免降低了，一般适用于模块中的函数比较少的时候，并且在程序中应用比较频繁的情况。

以上用math模块为例，引入其中的函数，其实在编程中，各样的对象都可以引入。

2.2.3　赋值

大家对于赋值语句应该不陌生，在前面已经频繁使用了，如a＝3这样的，就是将一个整数赋给了变量。

编程中的“＝”和数学中的“＝”是完全不同的。在编程语言中，“＝”表示赋值过程。

除了那种最简单的赋值之外，还可以这么做：

```
>>> x, y, z = 1, "python", ["hello", "world"]
>>> x
1
>>> y
'python'
>>> z
['hello', 'world']
```

这里就一一对应赋值了。如果把几个值赋给一个，可以吗？

```
>>> a = "itdiffer.com", "python"
>>> a
('itdiffer.com', 'python')
```

原来是将右边的两个值装入了一个元组，然后将元组赋给了变量a。Python太聪明了。

在Python的赋值语句中，还有更聪明的。

有两个变量，其中a＝2，b＝9。现在想让这两个变量的值对调，即最终是a＝9，b＝2。

这是一个简单而经典的题目。在很多编程语言中，是这样处理的：

```
temp = a;
a = b;
b = temp;
```

在这里变量就如同一个盒子，值就如同放到盒子里面的东西。如果要实现对调，必须再找一个盒子，将a盒子里面的东西（数字2）拿到那个临时盒子（temp）中，这样a盒子就空了，然后将b盒子中的东西（数字9）拿到a盒子中（a＝b），完成这步之后，b盒子是空的了，最后将临时盒子里面的那个数字2拿到b盒子中。这就实现了两个

变量值的对调。

Python只要一行就完成了。

```
>>> a = 2
>>> b = 9

>>> a, b = b, a

>>> a
9
>>> b
2
```

a，b＝b，a就实现了数值对调，多么神奇。之所以神奇，是因为前面已经数次提到的Python中变量和数据对象的关系。变量相当于贴在对象上的标签，这个操作只不过是将标签换个位置，就分别指向了不同的数据对象。

还有一种赋值方式，被称为“链式赋值”。

```
>>> m = n = "I use python"
>>> print m, n
I use python I use python
```

用这种方式实现了一次性对两个变量赋值，并且值相同。

```
>>> id(m)
3072659528L
>>> id(n)
3072659528L
```

用id()来检查一下，发现两个变量所指向的是同一个对象。

另外，还有一种判断方法，可以检查两个变量所指向的值是否是同一个（注意，同一个和相等是有差别的。在编程中，同一个就是id()的结果一样）。

```
>>> m is n
True
```

这是在检查m和n分别指向的对象是否是同一个，True说明是同一个。

```
>>> a = "I use python"
>>> b = a
>>> a is b
True
```

这跟上面的链式赋值是等效的。

但是：

```
>>> b = "I use python"
>>> a is b
False
>>> id(a)
3072659608L
>>> id(b)
3072659568L

>>> a == b
True
```

看出其中的端倪了吗？这次a、b两个变量虽然相等，但不是指向同一个对象。

还有一种赋值形式，如果从数学的角度看是不可思议的，如x＝x＋1，在数学中，这个等式是不成立的，因为数学中的“＝”是等于的含义，但是在编程语言中成立，因为“＝”是赋值的含义，即将变量x增加1之后，再把得到的结果赋给变量x。

这种变量自己变化之后将结果再赋值给自己的形式，称为“增量赋值”。＋、-、*、/、%都可以实现这种操作。

为了让这个操作写起来省点事儿，可以写成：x＋＝1，如

```
>>> x = 9
>>> x += 1
>>> x
10
```

除了数字，在实际中字符串进行增量赋值也很有价值。

```
>>> m = "py"
>>> m += "th"
>>> m
'pyth'
```

```
>>> m += "on"
>>> m
'python'
```

2.3 条件语句

条件，是日常生活中经常见到的，比如你给某个公司干活，条件是他支付薪水。所以，条件语句，也见诸各种高级编程语言。

2.3.1 if语句

if语句是由if发起的一个语句，即if发起是一个条件，在满足此条件后执行相应的内容。if这个单词就是构成条件语句的关键词。

```
>>> a = 8
>>> if a==8:
...     print a
...
8
```

在交互模式下，简单书写一下if发起的条件语句。特别说明，在交互模式中写稍微长一点的程序，本身不值得提倡，此处只是为了演示。如果你要写大段的代码，千万不要在交互模式下写。

if a= =8:这句话里面如果条件a= =8返回的是True，那么就执行下面的语句。此语句最后的冒号是必需的，下面一行语句print a要有四个空格的缩进。这是Python的特点，称之为语句块。

唯恐说得不严谨，我还是引用维基百科中的叙述：

Python开发者有意让违反了缩排规则的程序不能通过编译，以此来强迫程序员养成良好的编程习惯。并且Python语言利用缩排表示语句块的开始和结束（Off-side规则），而非使用花括号或者某种关键词。增加缩排表示语句块的开始，而减少缩排则表示语句块的结束。缩排成为了语法的一部分，例如if语句。

根据PEP的规定，必须使用四个空格来表示每级缩排。使用Tab字符和其他数目的空格虽然都可以编译通过，但不符合编码规范。支持Tab字符和其他数目的空格仅仅是为兼容很旧的Python程序和某些有问题的编辑程序。

从上面的这段话中，提炼出几个必需的要求：

- 必须要通过缩进方式来表示语句块的开始和结束。
- 缩进用四个空格（也是必需的，别的方式或许也可以，但不提倡）。

2.3.2 if…elif…else

在进行条件判断的时候，只有if往往是不够的。看如图2-1所示的流程。

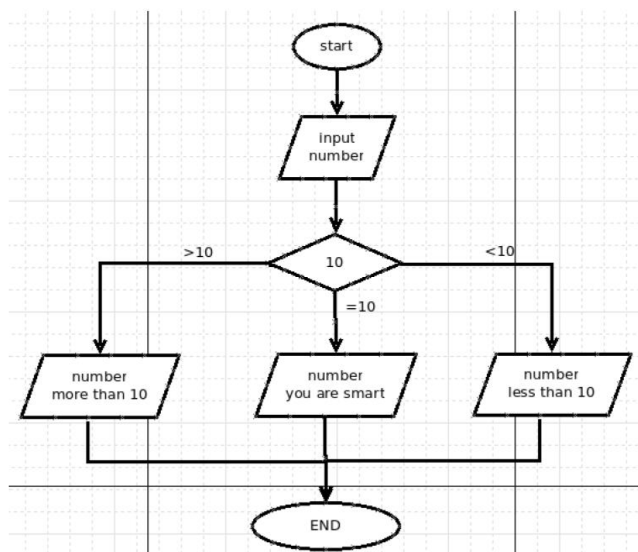


图2-1 流程

这张图反映的是这样一个问题：

输入一个数字，并输出输入的结果。如果这个数字大于10，那么输出提示大于10；如果小于10，则输出提示小于10；如果等于10，就输出表示表扬的一句话。

从图中就已经显示出来了，对一个值的判断结果有三个分支，为了解决多分支的问题，就引入另外一种条件判断：if...elif...else语句。

基本样式结构：

```
if 条件1:
    语句块1
elif 条件2:
    语句块2
elif 条件3：
    语句块3
else:
    语句块4
```

elif用于多个条件时，也可以没有。那就回归到if了。

下面就不在交互模式中写代码了。打开你的编辑器，代码实例如下：

```
#! /usr/bin/env python
#coding:utf-8

print "请输入任意一个整数数字："

number = int(raw_input())     #通过raw_input()输入的数字是字符串
                              #用int()将该字符串转化为整数

if number == 10:
    print "您输入的数字是：%d" % number
    print "You are SMART."
elif number > 10:
    print "您输入的数字是：%d" % number
    print "This number is more than 10."
elif number < 10:
    print "您输入的数字是：%d" % number
    print "This number is less than 10."
else:
    print "Are you a human?"
```

raw_input()函数获得用户在界面上输入的信息，而通过它得到的虽然是数字，但它是字符串类型的，所以要转化为整数型，才能用于后面的比较。

上述程序要依据条件进行判断，在不同条件下做不同的事情。需要提醒的是：在条件中，number＝＝10，为了阅读方便，在number和＝＝之间最好有一个空格，同理，后面也有一个。这里的10是int类型，number所引用的对象也是int类型。

把这段程序保存成一个扩展名是.py的文件，我把它保存为num.py，进入到存储这个文件的目录，运行python num.py，就能看到程序执行结果了。下面是我执行的结果，供参考。

```
$ python num.py
请输入任意一个整数数字：
12
您输入的数字是：12
This number is more than 10.

$ python num.py
请输入任意一个整数数字：
10
您输入的数字是：10
You are SMART.

$ python num.py
请输入任意一个整数数字：
9
您输入的数字是：9
This number is less than 10.
```

在“条件”中，就是前面已经提到的各种条件运算表达式，如果是True，就执行该条件下的语句。

2.3.3　三元操作符

三元操作，是条件语句中比较简练的一种赋值方式，它的模样是这样的：

```
>>> name = "qiwsir" if "laoqi" else "github"
>>> name
'qiwsir'
>>> name = 'qiwsir' if "" else "python"
>>> name
'python'
>>> name = "qiwsir" if "github" else ""
>>> name
'qiwsir'
```

总结一下：A＝Y if X else Z

结合前面的例子，可以看出：

- 如果X为真，那么就执行A＝Y。
- 如果X为假，就执行A＝Z。

如此例：

```
>>> x = 2
>>> y = 8
>>> a = "python" if x > y else "qiwsir"
>>> a
'qiwsir'
>>> b = "python" if x < y else "qiwsir"
>>> b
'python'
```

2.4 for循环

循环，也是现实生活中常见的现象，我们常说的日复一日就是典型的循环。又如日月更迭、斗转星移，无不是循环；王朝更迭、子子孙孙、繁衍不息，从某个角度看也都是循环。

编程语言就是要解决现实问题的，因此也少不了要循环。

在Python中，循环之一：for循环。言外之意，还有别的循环，请读者保持阅读的耐心，继续。

For循环的基本结构是：

```
for 循环规则：
    操作语句
```

从基本结构看，其有着同if条件语句类似的地方：都有冒号；语句块都要缩进。是的，这些是不可或缺的。

2.4.1 简单的for循环

前面介绍print语句的时候，出现了一个简单例子：

```
>>> hello = "world"
>>> for i in hello:
...     print i
...
w
o
r
l
d
```

这个for循环是怎么工作的呢？

- hello这个变量引用的是“world”这个字符串类型的数据。
- 变量i通过“hello”找到它所引用的对象“world”，因为字符串类

型的对象属于序列类型，能够进行索引，于是就按照索引顺序，从第一字符开始，依次获得该字符。
- 当i＝"w"的时候，执行print i，打印出了字母w，结束之后循环第二次，让i＝"e"，然后执行print i，打印出字母e。如此循环下去，一直到最后一个字符被打印出来，循环自动结束。注意，每次打印之后要换行。

因为也可以通过使用索引（偏移量）得到序列对象的某个元素，所以，还可以通过下面的循环方式实现同样的效果：

```
>>> for i in range(len(hello)):
...     print hello[i]
...
w
o
r
l
d
```

其工作方式是：

- len（hello）得到hello引用的字符串的长度，为5。
- range（len（hello）就是range（5），也就是[0，1，2，3，4]，对应着“world”每个字母索引，也可以称之为偏移量。这里应用了一个新的函数range()，关于它的用法，继续阅读就能看到了。
- for i in range（len（hello））就相当于for i in[0，1，2，3，4]，让i依次等于list中的各个值。当i＝0时，打印hello[0]，也就是第一个字符。然后顺序循环下去，直到最后一个i＝4为止。

以上的循环举例中，显示了对字符串的字符依次获取，同时涉及了列表，再看下面对列表的循环：

```
>>> ls_line
['Hello', 'I am qiwsir', 'Welcome you', '']
>>> for word in ls_line:
...     print word
...
Hello
I am qiwsir
Welcome you

>>> for i in range(len(ls_line)):
```

```
...     print ls_line[i]
...
Hello
I am qiwsir
Welcome you
```

2.4.2 range（start，stop[，step]）

range()是个内建函数，一般形式是range（start，stop[，step]）。

要研究清楚一些函数()特别是内置函数()的功能，建议读者首先要明白内置函数名称的含义。因为在Python中，名称不是随便取的，是代表一定意义的。

在具体实验之前，还是按照惯例，摘抄一段官方文档的原话，让我们能够深刻理解之：

This is a versatile function to create lists containing arithmetic progressions.It is most often used in for loops.The arguments must be plain integers.If the step argument is omitted，it defaults to 1.If the start argument is omitted，it defaults to 0.The full form returns a list of plain integers[start，start＋step，start＋2*step，...].If step is positive，the last element is the largest start＋i*step less than stop;if step is negative，the last element is the smallest start＋i*step greater than stop.step must not be zero（or else ValueError is raised）.

关于range()函数注意以下几点：

- 这个函数可以创建一个数字元素组成的列表。
- 这个函数最常用于for循环。
- 函数的参数必须是整数，默认从0开始。返回值是类似[start，start＋step，start＋2*step，...]的列表。
- step默认值是1。如果不写，就是按照此值。
- 如果step是正数，返回list的最后的值不包含stop值，即start＋istep这个值小于stop；如果step是负数，start＋istep的值大于stop。
- step不能等于零，如果等于零，就报错。

再对各个参数给予详细解释。

- start：开始数值，默认为0，即如果不写这项，就是认为start＝0。
- stop：结束的数值，必须要写。
- step：变化的步长，默认是1，即若不写则认为步长为1，坚决不能为0。

实验开始，并对照前面所讲述的参数含义：

```
>>> range(9)                    #stop=9，别的都没有写，含义就是range(0, 9,
[0, 1, 2, 3, 4, 5, 6, 7, 8]              #从0开始，步长为1，直到小于9的那个
>>> range(0, 9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> range(0, 9, 1)
[0, 1, 2, 3, 4, 5, 6, 7, 8]

>>> range(1, 9)                 #start=1
[1, 2, 3, 4, 5, 6, 7, 8]

>>> range(0, 9, 2)              #step=2,每个元素等于start+i*step
[0, 2, 4, 6, 8]
```

仅仅解释一下range（0，9，2）：

- 如果是从0开始，步长为1，可以写成range（9）的样子，但是，如果步长为2，写成range（9，2）的样子，计算机就有点糊涂了，它会认为start＝9，stop＝2。所以，在步长不为1的时候，一定要把start的值也写上。
- start＝0，step＝2，stop＝9。返回的列表中的第一个值是start＝0，第二个值是start＋1*step＝2（注意，这里是1，不是2，不论是列表还是字符串，索引值都是从0开始的），第n个值就是start＋（n-1）*step。直到小于stop前的那个值。

熟悉了上面的计算过程，想一想这个的结果是什么。

```
>>> range(-9)
```

期望返回[0，-1，-2，-3，-4，-5，-6，-7，-8]能实现吗？

分析一下，这里start＝0，step＝1，stop＝-9。

第一个值是0；第二个是start＋1*step，将上面的数代入应该是

1，但是最后是-9，显然出现问题了。但是，Python在这里不报错，它返回的结果是：

```
>>> range(-9)
[]
>>> range(0, -9)
[]
>>> range(0)
[]
```

报错和返回结果是两个含义。

返回的不是我们要的。应该如何修改呢？

```
>>> range(0, -9, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8]
>>> range(0, -9, -2)
[0, -2, -4, -6, -8]
```

有了这个内置函数，很多事情就简单了。比如：

```
>>> range(0, 100, 2)
    [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32
```

100以内的自然数中的偶数组成的列表就非常简单地搞定了上面那个问题。

思考一个问题，现在有一个列表，比如是["I"，"am"，"a"，"pythoner"，"I"，"am"，"learning"，"it"，"with"，"qiwsir"]，要得到这个列表的索引值组成的列表，但是不能一个一个用手指头来数。怎么办？

请沉思两分钟之后，自己实验一下，然后看下面。

```
>>> pythoner
['I', 'am', 'a', 'pythoner', 'I', 'am', 'learning', 'it', 'with',
>>> py_index = range(len(pythoner))            #以len(pythoner)为
>>> py_index
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

再用手指头指着Pythoner里面的元素数一数，是不是跟结果一样？

例：找出100以内的能够被3整除的正整数。

分析：这个问题有两个限制条件，第一是100以内的正整数，根据前面所学，可以用range（1，100）来实现；第二个是要解决被3整除的问题，假设某个正整数n能够被3整除，也就是n%3（%是取余数）为0。那么如何得到n呢，就是要用for循环。

以上做了简单分析，要实现流程，还需要细化一下。按照前面曾经讲授过的一种方法，要画出解决问题的流程图，如图2-2所示。

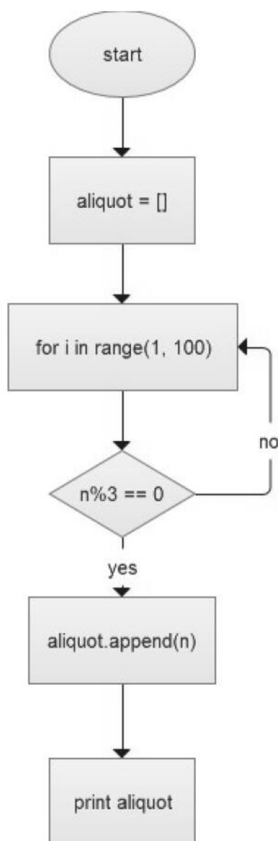


图2-2　解决问题的流程图

下面写代码就是“按图索骥”了。

```
#! /usr/bin/env python
#coding:utf-8
```

```
aliquot = []

for n in range(1, 100):
    if n % 3 == 0:
        aliquot.append(n)

print aliquot
```

代码运行结果：

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,
```

在上面的代码中，将for循环和if条件判断都用上了。

不过，感觉有点儿麻烦，其实这么做就可以了：

```
>>> range(3,100,3)
    [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48,
```

2.4.3 for的对象

所有的序列类型对象都能够用for来循环。比如：

```
>>> name_str = "qiwsir"
>>> for i in name_str:
...     print i,
...
q i w s i r

>>> name_list = list(name_str)
>>> name_list
['q', 'i', 'w', 's', 'i', 'r']
>>> for i in name_list:
...     print i,
...
q i w s i r

>>> name_set = set(name_str)     #set还可以用
>>> name_set
set(['q', 'i', 's', 'r', 'w'])
>>> for i in name_set:
...     print i,
...
q i s r w
```

```
>>> name_tuple = tuple(name_str)
>>> name_tuple
('q', 'i', 'w', 's', 'i', 'r')
>>> for i in name_tuple:        #tuple也能呀
...     print i,
...
q i w s i r

>>> name_dict={"name": "qiwsir", "lang": "python", "website": "qiw
>>> for i in name_dict:             #dict也不例外
...     print i, "-->", name_dict[i]
...
lang --> python
website --> qiwsir.github.io
name --> qiwsir
```

用for循环读取字典"键/值"对需要多说几句。

有这样一个字典：

```
>>> a_dict = {"name": "qiwsir", "lang": "python", "email": "qiwsir
```

获得字典键、值的函数有：items/iteritems/keys/iterkeys/values/itervalues，通过这些函数得到的是键或者值的列表。

```
>>> for k in a_dict():
...     print k, a_dict[k]
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com
```

这是一种获得字典键/值对的方法，通常情况下较常用，效率也能满足一般需要。

```
>>> for k,v in a_dict.items():
...     print k, v
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com

>>> for k,v in a_dict.iteritems():
```

```
...     print k, v
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com
```

这两种方法也能够实现同样的效果，特别是第二个iteritems()效率挺高。

但是，要注意下面的方法：

```
>>> for k in a_dict.keys():
...     print k, a_dict[k]
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com
```

这种方法所达到的效果跟前面一样，但不太提倡，因为效率比较低。

```
>>> for v in a_dict.values():
...     print v
...
python
www.itdiffer.com
qiwsir
qiwsir@gmail.com

>>> for v in a_dict.itervalues():
...     print v
...
python
www.itdiffer.com
qiwsir
qiwsir@gmail.com
```

单独取values，推荐第二种方法。

2.4.4 zip()

前面了解了“可迭代的（iterable）”这个词，这里再次提到“迭代”，说明它在Python中占有重要的位置。

迭代在Python中的表现就是用for循环，从序列对象中获得一定数量的元素。

用for循环来获得列表、字符串、元组，乃至于字典的键/值对都是迭代。

现实中迭代不是都那么简单的，比如下面这个问题。

问题：有两个列表，分别是：a＝[1，2，3，4，5]，b＝[9，8，7，6，5]，要计算这两个列表中对应元素的和。

解析：

经观察发现两个列表的长度一样，都是5。那么对应元素求和，就是相同的索引值对应的元素求和，即a[i]＋b[i]（i＝0，1，2，3，4），这样一个一个地就把相应元素和求出来了。当然，要用for来做这个事情了。

```
>>> a = [1,2,3,4,5]
>>> b = [9,8,7,6,5]
>>> c = []
>>> for i in range(len(a)):
...     c.append(a[i] + b[i])
...
>>> c
[10, 10, 10, 10, 10]
```

看来for的表现还不错。

这种方法虽然解决了问题，但Python总不会局限于一个解决之道。于是又有一个内建函数zip()，可以让同样的问题有不一样的解决途径。

zip()是什么东西？在交互模式下用help（zip），得到官方文档是：

zip(...)zip(seq1[,seq2[...]])-＞ [(seq1[0],seq2[0]...),(...)]

Return a list of tuples,where each tuple contains the i-th element from each of the argument sequences.The returned list is

truncated in length to the length of the shortest argument sequence.

seq1、seq2分别代表了序列类型的数据。通过实验来理解上面的文档：

```
>>> a = "qiwsir"
>>> b = "github"
>>> zip(a,b)
[('q', 'g'), ('i', 'i'), ('w', 't'), ('s', 'h'), ('i', 'u'), ('r',
```

如果序列长度不同，那么就以“the length of the shortest argument sequence”为准。

```
>>> c = [1,2,3]
>>> d = [9,8,7,6]
>>> zip(c,d)
[(1, 9), (2, 8), (3, 7)]

>>> m = {"name","lang"}
>>> n = {"qiwsir","python"}
>>> zip(m,n)
[('lang', 'python'), ('name', 'qiwsir')]
```

m、n是字典吗？当然不是，下面的才是字典呢。

```
>>> s = {"name":"qiwsir"}
>>> t = {"lang":"python"}
>>> zip(s,t)
[('name', 'lang')]
```

zip()是一个内置函数，它的参数必须是某种序列数据类型，如果是字典，那么视为序列。然后将序列对应的元素依次组成元组，并单做列表中的元素。

下面是比较特殊的情况，当参数是一个序列时所生成的结果：

```
>>> a
'qiwsir'
>>> c
[1, 2, 3]
>>> zip(c)
[(1,), (2,), (3,)]
>>> zip(a)
[('q',), ('i',), ('w',), ('s',), ('i',), ('r',)]
```

其实也不特殊，因为只提供了一个参数，那么列表中的元组就一个元素，此时元组中元素后面要有一个逗号（半角的）。

很好的zip()！那么就用它来解决前面列表中值对应相加问题吧。

```
>>> d = []
>>> for x,y in zip(a,b):
...     d.append(x + y)
...
>>> d
[10, 10, 10, 10, 10]
```

比较这个问题的两种解法，似乎第一种解法适用面较窄，比如，如果已知给定的两个列表长度不同，那么第一种解法就出问题了，而第二种解法还可以继续适用。的确如此，不过，第一种解法也不是不能修订的。

```
>>> a = [1,2,3,4,5]
>>> b = ["python","www.itdiffer.com","qiwsir"]
```

如果已知是这样两个列表，要将对应的元素“加起来”。

```
>>> length = len(a) if len(a)<len(b) else len(b)
>>> length
3
```

首先用这种方法获得两个列表中最短的那个列表的长度。写出这句，就可以冒充高手了。

```
>>> for i in range(length):
...     c.append(str(a[i]) + ":" + b[i])
...
>>> c
['1:python', '2:www.itdiffer.com', '3:qiwsir']
```

我还是用第一个思路做的，经过这么修正一下，也还能用。要注意一个细节，在“加”的时候，不能直接用a[i]，因为它引用的对象是一个int类型，不能跟后面的str类型相加，必须转化一下。

当然，zip()也是能解决这个问题的。

```
>>> d = []
```

```
>>> for x,y in zip(a,b):
...     d.append(x + y)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

报错！看错误信息，我刚刚提醒的那个问题就冒出来了。所以，应该这么做：

```
>>> for x,y in zip(a,b):
...     d.append(str(x) + ":" + y)
...
>>> d
['1:python', '2:www.itdiffer.com', '3:qiwsir']
```

这才得到了正确结果。

切记：Computer是一个姑娘，她非常秀气，需要敲代码的小伙子们耐心地、细心地跟她相处。

以上两种写法哪个更好呢？

```
>>> result
[(2, 11), (4, 13), (6, 15), (8, 17)]
>>> zip(*result)
[(2, 4, 6, 8), (11, 13, 15, 17)]
```

zip()还能这么干，是不是有点意思？

下面延伸一个问题。

问题：有一个字典yinfor＝{"name":"qiwsir"，"site":"qiwsir.github.io"，"lang":"python"}，这个字典变换成：infor＝{"qiwsir":"name"，"qiwsir.github.io":"site"，"python":"lang"}。

解析：

解法有几个，如果用for循环，可以这样做（当然，方法不限于下列几个）。

```
>>> infor = {}
```

```
>>> for k, v in myinfor.items():
...     infor[v]=k
...
>>> infor
{'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

下面用zip()来试试：

```
>>> dict(zip(myinfor.values(), myinfor.keys()))
{'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

这是什么情况？原来这个zip()还能这样用。是的，本质上是这么回事。如果将上面这一行分解开来，就明白其中的奥妙了。

```
>>> myinfor.values()
['python', 'qiwsir', 'qiwsir.github.io']
>>> myinfor.keys()
['lang', 'name', 'site']
>>> temp = zip(myinfor.values(),myinfor.keys())
>>> temp
[('python', 'lang'), ('qiwsir', 'name'), ('qiwsir.github.io', 'sit

>>> dict(temp)
{'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

至此，是不是明白zip()和循环的关系了呢？有了它可以让某些循环简化。

2.4.5 enumerate()

本来可以通过for i in range（len（list））的方式得到一个list的每个元素索引，然后再用list[i]的方式得到该元素。如果要同时得到元素索引和元素怎么办？可以这样：

```
>>> for i in range(len(week)):
...     print week[i]+' is '+str(i)
...
monday is 0
sunday is 1
friday is 2
```

注意，i是int类型，如果和前面的用+连接，必须是str类型。

Python中提供了一个内置函数enumerate，能够实现类似的功能。

```
>>> for (i,day) in enumerate(week):
...     print day+' is '+str(i)
...
monday is 0
sunday is 1
friday is 2
```

官方文档是这么说的：

Return an enumerate object.sequence must be a sequence，an iterator，or some other object which supports iteration.The next()method of the iterator returned by enumerate()returns a tuple containing a count（from start which defaults to 0）and the values obtained from iterating over sequence:

顺便抄录几个例子，供读者欣赏，但最好自己实验一下。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

对于这样一个列表：

```
>>> mylist = ["qiwsir",703,"python"]
>>> enumerate(mylist)
<enumerate object at 0xb74a63c4>
```

出现这个结果，用list就能实现转换，显示内容意味着可迭代。

```
>>> list(enumerate(mylist))
[(0, 'qiwsir'), (1, 703), (2, 'python')]
```

再设计一个小问题，练习一下这个函数。

问题：将字符串中的某些字符替换为其他的字符串。原始字符串为"Do you love Canglaoshi?Canglaoshi is a good teacher."，请将"Canglaoshi"替换为"PHP"。

解析：

```
>>> raw = "Do you love Canglaoshi? Canglaoshi is a good teacher."
```

不能直接对这个字符串使用enumerate()，因为它会变成这样：

```
>>> list(enumerate(raw))
    [(0, 'D'), (1, 'o'), (2, ' '), (3, 'y'), (4, 'o'), (5, 'u'), (
```

这不是所需要的，所以，先把raw转化为列表：

```
>>> raw_lst = raw.split(" ")
```

然后用enumerate()：

```
>>> for i, string in enumerate(raw_lst):
...     if string == "Canglaoshi":
...         raw_lst[i] = "PHP"
```

没有什么异常现象，查看一下那个raw_lst列表，看看是不是把"Canglaoshi"替换为"PHP"了。

```
>>> raw_lst
['Do', 'you', 'love', 'Canglaoshi?', 'PHP', 'is', 'a', 'good', 'te
```

只替换了一个，还有一个没有替换，为什么？仔细观察发现，没有替换的那个是'Canglaoshi?'，跟条件判断中的"Canglaoshi"不一样。

修改一下，把条件放宽：

```
>>> for i, string in enumerate(raw_lst):
...     if "Canglaoshi" in string:
...         raw_lst[i] = "PHP"
...
>>> raw_lst
['Do', 'you', 'love', 'PHP', 'PHP', 'is', 'a', 'good', 'teacher.']
```

然后再转化为字符串，留给读者试试。

2.4.6 列表解析

先看下面的例子，想得到1到9每个整数的平方，并且将结果放在列表中打印出来。

```
>>> power2 = []
>>> for i in range(1, 10):
...     power2.append(i*i)
...
>>> power2
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python有一个非常有意思的功能，就是list解析，是这样的：

```
>>> squares = [x**2 for x in range(1,10)]
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

看到这个结果还不惊叹吗？这就是Python，追求简洁优雅的Python！

其官方文档中有这样一段描述，道出了list解析的真谛：

List comprehensions provide a concise way to create lists.Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable，or to create a subsequence of those elements that satisfy a certain condition.

这就是Python有意思的地方，也是计算机高级语言编程有意思的地方，你只要动脑筋，就能找到惊喜。

请在平复了激动的心之后，默默地看下面的代码，感悟一下list解析的魅力。

```
>>> mybag = [' glass',' apple','green leaf ']    #有的前面有空格，有的
>>> [one.strip() for one in mybag]               #去掉元素前后的空格
['glass', 'apple', 'green leaf']
```

list解析的执行效率高，代码简洁明了，在实际写程序中经常会用到。

2.5 while循环

while，翻译成中文是“当……的时候”，这个单词在英语中常常用来作为时间状语，while...someone do somthing，这种类型的说法是有的。在Python中，它也有这个含义，区别是“当……时候”这个条件成立在一段范围或者时间间隔内，从而在这段时间间隔内让Python做好多事情。就好比这样一段情景：

```
while 年龄大于60岁：                #当年龄大于60岁的时候
      退休                          #凡是符合上述条件就执行的动作
```

展开想象，如果制作一道门，这道门是用上述的条件调控开关，假设有很多人经过这个门，只要年龄大于60岁就退休（门打开，人可以出去），一个接一个地这样循环下去，突然有一个人年龄是50岁，那么这个循环在这里就停止，即不满足条件了。

这就是while循环。写一个严肃点儿的流程，如图2-3所示。

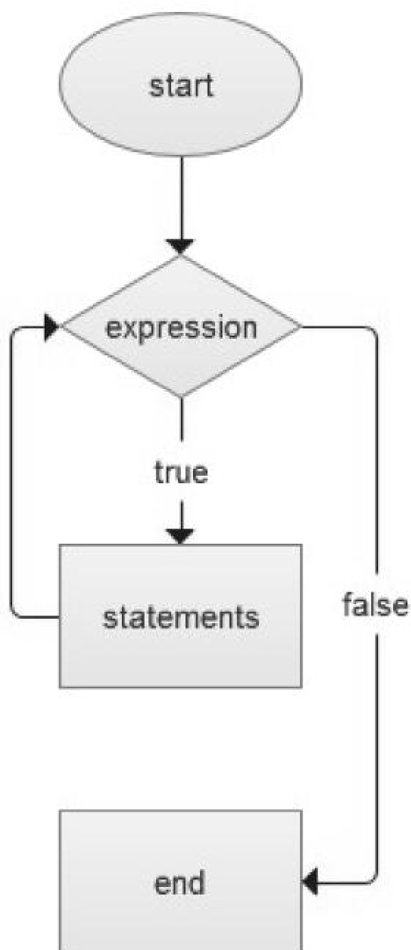


图2-3　while循环

2.5.1　猜数字游戏

前不久，有一个在校的大学生朋友（李航）给我发邮件，让我看了他做的游戏，这款游戏能够实现多次猜数，直到猜中为止。这是一个多么喜欢学习的大学生呀。

现在将他写的程序恭录于此，这一举动已经得到了李航同学在QQ上的授权，感谢他对本书的支持。

```
#! /usr/bin/env python
#coding:UTF-8

import random

i=0
while i < 4:
    print'********************************'
    num = input('请您输入0到9任一个数：')        #李同学用的是Python3

    xnum = random.randint(0, 9)

    x = 3 - i

    if num == xnum:
        print'运气真好，您猜对了！'
        break
    elif num > xnum:
        print'''您猜大了!\n哈哈,正确答案是:%s\n您还有%s次机会！''' %(xnu
    elif num < xnum:
        print'''您猜小了!\n哈哈,正确答案是:%s\n您还有%s次机会！''' %(xnu
    print'********************************'

    i += 1
```

我们来分析一下。

首先看while i＜4，这是程序中为猜测限制的次数，最多是三次，请注意，在while循环体中的最后一句：i+＝1，这是说每次循环到最后就给i增加1，当bool（I＜4）为False的时候，就不再循环了。

当bool（i＜4）为True的时候，就执行循环体内的语句。在循环体内，让用户输入一个整数，然后程序随机选择一个整数，最后判断随机生成的数和用户输入的数是否相等，并且用if语句判断三种不同的情况。

明白了上述代码，就可以进一步研究是否可以优化或者进行其他修改。

为了让用户的体验更爽，不妨把输入的整数范围扩大至1到100之间。

```
num_input = raw_input("please input one integer that is in 1 to 10
```

我用的是Python 2.7，在输入指令上区别于李同学。

程序用num_input变量接收了输入的内容。读者如果要在这里睡觉，请打起精神，我要分享一个多年的编程经验。

请牢记：任何用户输入的内容都是不可靠的。

这句话含义深刻，但是，这里不做过多的解释，需要各位在随后的编程生涯中体验。根据此经验，我们要检验用户输入的是否符合我们的要求，我们要求用户输入1到100之间的整数，那么要做如下检验：

- 输入的是否是整数。
- 如果是整数，是否在1到100之间。

为此，要做：

```
if not num_input.isdigit():          #str.isdigit()是用来判断字符串是
    print "Please input interger."
elif int(num_input)<0 and int(num_input)>=100:
    print "The number should be in 1 to 100."
else:
    pass
```

这里用pass的意思是暂时省略，如果满足了前面提出的要求，就该执行此处语句。

再看看李航同学的程序，在循环体内产生一个随机的数字，这样用户每次输入，面对的都是一个新的随机数字，这使得猜数字游戏难度太大了。我希望程序产生一个数字，直到猜中都是这个数字。所以，要把产生随机数字这个指令移动到循环之前。

```
import random

number = random.randint(1,100)

while True:                #不限制用户的次数了
    ...
```

观察李同学的程序，还有一点需要说明，那就是在条件表达式中，两边最好是同种类型数据，上面的程序中有num＞xnum样式的条件表达式，一边是程序生成的int类型数据，而另一边是通过输入函数得到的str类型数据。在某些情况下可以运行，为什么？都是数字的时候是可以的，但是这样不好。

那么，按照这种思路，把这个猜数字程序重写一下：

```
#!/usr/bin/env python
#coding:utf-8

import random

number = random.randint(1,101)

guess = 0

while True:

    num_input = raw_input("please input one integer that is in 1 t
    guess += 1

    if not num_input.isdigit():
        print "Please input interger."
    elif int(num_input) < 0 or int(num_input) >= 100:
        print "The number should be in 1 to 100."
    else:
        if number == int(num_input):
            print "OK, you are good.It is only %d, then you succes
            break
        elif number > int(num_input):
            print "your number is more less."
        elif number < int(num_input):
            print "your number is bigger."
        else:
        print "There is something bad, I will not work"
```

以上仅供参考，读者还可改进。

2.5.2 break和continue

break的含义就是要在这个地方中断循环，跳出循环体。下面这个简要的例子更明显：

```
#!/usr/bin/env python
#coding:utf-8

a = 8
while a:
    if a % 2 == 0:
        break
```

```
    else:
        print "%d is odd number"%a
        a = 0
print "%d is even number"%a
```

a＝8的时候，执行循环体中的break跳出循环，执行最后的打印语句，得到结果：

```
8 is even number
```

如果a＝9，则要执行else里面的print，然后a＝0，循环就再执行一次，又break了，得到结果：

```
9 is odd number
0 is even number
```

而continue则是要从当前位置（即continue所在的位置）跳到循环体的最后一行的后面（不执行最后一行），对一个循环体来讲，就如同首尾衔接一样，最后一行的后面是哪里呢？当然是开始了。

```
#!/usr/bin/env python
#coding:utf-8

a = 9
while a:
    if a % 2 == 0:
        a -= 1
        continue                            #如果是偶数，就返回循环的开始
    else:
        print "%d is odd number"%a          #如果是奇数，就打印出来
        a -= 1
```

其实，对于这两个条件，我个人在编程中很少用到，因为我有一个固执的观念，尽量将条件在循环之前做足，不要在循环中跳来跳去，因为这样不仅可读性下降了，有时候自己也容易糊涂。

2.5.3 while...else

while...else有点类似于if...else，只需要一个例子就理解了，当然，一遇到else，就意味着已经不在while循环内了。

```
#!/usr/bin/env python

count = 0
while count < 5:
    print count, " is  less than 5"
    count = count + 1
else:
    print count, " is not less than 5"
```

执行结果：

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

2.5.4 for...else

除了有while...else外，还可以有for...else，这个循环也通常用于跳出循环之后要做的事情。

```
#!/usr/bin/env python
# coding=utf-8

from math import sqrt

for n in range(99, 1, -1):
    root = sqrt(n)
    if root == int(root):
        print n
        break
else:
    print "Nothing."
```

读者是否能够读懂这段代码的含义？

阅读代码是一个提升自己编程水平的好方法。如何阅读代码？像看网上新闻那样吗？一目只看自己喜欢的文字，甚至标题看不完就开始喷？

绝对不是这样，阅读代码的最好方法是给代码做注释。对，如果

有可能就给每行代码做注释，这样就能理解代码的含义了。

对于上面的代码，读者不妨做注释，看看它到底在干什么。把for n in range（99，1，-1）修改为for n in range（99，81，-1）看看是什么结果。

2.6 文件

文件是Computer姑娘非常重要的东西，在Python里，它也是一种类型的对象，类似前面已经学习过的其他类型，包括文本的、图片的、音频的、视频的等，还有不少没见过的扩展名的。事实上，在Linux操作系统中，所有的东西都被保存到文件中。

先在交互模式下查看文件都有哪些属性和函数：

```
>>> dir(file)
    ['__class__', '__delattr__', '__doc__', '__enter__', '__exit__
```

这里只对部分属性或函数进行详细说明，读者可以用本书一贯倡导的方法自学。在网上看到过一个统计，绝大多数程序员都是自学成才的，你可以问问周围的大牛程序员，自学在他们的职业生涯中占据什么地位。

特别注意观察，在上面有_iter_这个东西。在讲述列表的时候它是不是也曾经出现了呢？如果没有注意看，请自己在交互模式中找出来。_iter_是一个重要的标志，它意味着这种类型的数据是可迭代的。接下来，你就会看到，可迭代的是多么神奇。

2.6.1 打开文件

在某个文件夹下面建立了一个文件，名为：130.txt，并且在里面输入了如下内容：

```
learn python
http://qiwsir.github.io
qiwsir@gmail.com
```

此文件一共三行。

下图显示了这个文件的存储位置：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105-1.py  106-1.py  111-1.py  129-1.py  130.txt
105.py    109.py    118-1.py  129-2.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python
```

我在当前位置输入了python（我已经设置了环境变量，如果你没有设置，需要写全启动Python命令路径），进入到交互模式。在交互模式下，这样操作：

```
>>> f = open("130.txt")      #打开已经存在的文件
>>> for line in f:
...     print line
...
learn python

http://qiwsir.github.io

qiwsir@gmail.com
```

将打开的文件赋值给变量f，这样就是变量f跟对象文件130.txt用线连起来了（对象引用），本质上跟前面所讲述的其他类型数据进行赋值是一样的。

接下来，用for来读取文件中的内容，就如同读取一个前面已经学过的序列对象一样，如列表、字符串、元组，将读到的文件中的每一行赋值给变量line。也可以理解为，for循环是一行一行地读取文件内容。每次扫描一行，遇到行结束符号\n表示本行结束，然后是下一行。

从打印的结果可以看出，打印的每一行内容跟文件中每一行的内容是一样的，只是行与行之间多了个空行，前面显示文章内容的时候，并没有这个空行。或许这无关紧要，但是，还要深究一下才能豁然。

在原文中，每行结束都有结束符号\n，表示换行。在for语句汇总，print line表示每次打印完line的对象之后就换行，也就是打印完line的对象之后会增加一个\n。这样看来，在每行末尾就有两个\n，即：\n\n，于是在打印中就出现了一个空行。

```
>>> f = open('130.txt')
>>> for line in f:
...     print line,     #后面加一个逗号，就去掉了原来默认增加的\n了
...
learn python
http://qiwsir.github.io

```

```
qiwsir@gmail.com
```

在进行上述操作的时候，有没有遇到这样的情况呢？

```
>>> f = open('130.txt')
>>> for line in f:
...     print line,
...
learn python
http://qiwsir.github.io
qiwsir@gmail.com

>>> for line2 in f:        #在前面通过for循环读取了文件内容之后，再次读取
...     print line2                    #然后打印，却没有显示任何结果
...
>>>
```

如果读者没有遇到上面的问题，可以试试。

这不是什么错误，是因为前一次已经读取了文件内容，并且到了文件的末尾了。再重复操作，就是从末尾开始继续读了。当然显示不了什么东西，但是Python并不认为这是错误，因为或许在这次读取之前，已经又向文件中追加内容了。那么，如果要再次读取怎么办？就重新来一遍好了。这就好比有一个指针在指着文件中的每一行，每读完一行，指针就移动一行。直到指针指向了文件的最末尾。当然，也有办法把指针移动到任何位置。

提醒读者注意，因为当前的交互模式是在该文件所在目录启动的，所以，就相当于这个实验室和文件130.txt是同一个目录，这时候我们打开文件130.txt，就认为是在本目录中打开，如果文件不是在本目录中，需要写清楚路径。

比如：在上一级目录中（~/Documents/ITArticles/BasicPython），假如进入到那个目录，运行交互模式，然后试图打开130.txt文件。

```
~/Documents/ITArticles/BasicPython/codes$ ls
11-1.py   129-1.py   130.txt
18-1.py   129-2.py
~/Documents/ITArticles/BasicPython/codes$ cd ..
~/Documents/ITArticles/BasicPython$ python
```

```
>>> f = open("130.txt")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '130.txt'
>>> f = open("./codes/130.txt")      #必须得写上路径了
>>> for line in f:
...     print line
...
learn python

http://qiwsir.github.io

qiwsir@gmail.com

>>>
```

2.6.2 创建文件

上面的实验中，打开的是一个已经存在的文件。如何创建一个新文件呢？

```
>>> nf = open("131.txt", "w")
>>> nf.write("This is a file")
```

这样就创建了一个文件吗？还写入了文件内容吗？

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105-1.py  106-1.py  111-1.py  129-1.py  130.txt
105.py    109.py    118-1.py  129-2.py  131.txt
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ cat 131.txt
This is a fileqw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$
```

从上图中可以看到，真的就创建了新文件，并且里面有“This is a file”那句话。

注意，这次同样是用open()这个函数，但是多了个“w”，这是在告诉Python用什么样的模式打开文件。也就是说，用open()打开文件，可以有不同的打开模式。如表2-1所示。

表2-1 打开模式

模式	描　述
r	以读方式打开文件，可读取文件信息
w	以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容
a	以追加模式打开文件（打开文件，文件指针自动移到文件末尾），如果文件不存在则创建
r+	以读写方式打开文件，可对文件进行读和写操作
w+	消除文件内容，然后以读写方式打开文件
a+	以读写方式打开文件，并把文件指针移到文件尾
b	以二进制模式打开文件，而不是以文本模式。该模式只对 Windows 或 DOS 有效，类 UNIX 的文件是用二进制模式进行操作的

以二进制模式打开文件，而不是以文本模式。该模式只对Windows或DOS有效，类UNIX的文件是用二进制模式进行操作的

从表2-1中不难看出，在不同模式下打开文件，可以进行相关的读写。那么，如果什么模式都不写，像前面那样呢？那样就默认为是r模式，以只读的方式打开文件。

```
>>> f = open("130.txt")
>>> f
<open file '130.txt', mode 'r' at 0xb7530230>
>>> f = open("130.txt","r")
>>> f
<open file '130.txt', mode 'r' at 0xb750a700>
```

可以用这种方式查看当前打开的文件是采用什么模式打开的，上面显示两种模式是一样的效果，如果不写那个"r"，就默认为是只读模式了。下面逐个对两个常用模式进行解释。

1."w"：以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容

131.txt这个文件是已经存在的，并且在里面有了一句话：This is a file。

```
>>> fp = open("131.txt")
>>> for line in fp:
...     print line
...
This is a file
>>> fp = open("131.txt","w")
>>> fp.write("My name is qiwsir.\nMy website is qiwsir.github.io")
>>> fp.close()
```

查看文件内容（cat是Linux下显示文件内容的命令，这里就是要显示131.txt内容）：

```
$ cat 131.txt
My name is qiwsir.
My website is qiwsir.github.io
```

2."a"：以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建。

```
>>> fp = open("131.txt","a")
>>> fp.write("\nAha,I like program\n")     #向文件中追加
>>> fp.close()
```

在上面两个写入的操作之后，都用close()来关闭文件，这是很重要的一步，一定要养成一个习惯，写完内容之后就关闭。

同样，查看文件内容：

```
$ cat 131.txt
My name is qiwsir.
My website is qiwsir.github.io
Aha,I like program
```

其他项目就不一一讲述了，读者可以自己实验。

2.6.3 使用with

在对文件进行写入操作之后，一定要牢记一件事情：file.close()，这个操作千万不要忘记，若忘记了那就补上，也没有什么天塌地陷的后果。

有另外一种方法能够不用这么让人揪心，实现安全地关闭文件。

```
>>> with open("130.txt","a") as f:
...     f.write("\nThis is about 'with...as...'")
...
>>> with open("130.txt","r") as f:
...     print f.read()
...
learn python
http://qiwsir.github.io
qiwsir@gmail.com
hello
```

```
This is about 'with...as...'
>>>
```

这里就不用close()了，而且这种方法更有Python味道，或者说更符合Pythonic的一个要求。

2.6.4 文件的状态

有时候需要知道一个文件的有关状态（也称为属性），比如创建日期、访问日期、修改日期、大小等。在os模块中，有这样一个方法，专门让我们查看文件的这些状态参数。

```
>>> import os
>>> file_stat = os.stat("131.txt")     #查看这个文件的状态
>>> file_stat
    posix.stat_result(st_mode=33204, st_ino=5772566L, st_dev=2049L

>>> file_stat.st_ctime                  #这是文件创建时间
1407734600.0882277
```

这是什么时间？看不懂！别着急，换一种方式。在Python中，有一个模块time，是专门针对时间设计的。

```
>>> import time
>>> time.localtime(file_stat.st_ctime)
    time.struct_time(tm_year=2014, tm_mon=8, tm_mday=11, tm_hour=1
```

2.6.5 read/readline/readlines

简单的读取文件内容已经了解过了，但是，在用dir（file）的时候会看到三个函数：read/readline/readlines，它们各自有什么特点？为什么是三个函数？

在读者看下面的内容之前，请想一想，如果要回答这个问题，你要用什么方法？注意，我问的是用什么方法能够找到答案，不是问答案内容是什么。因为内容肯定在某个地方存放着呢，关键是用什么方法找到。

搜索是一个不错的方法。

还有一种，就是在交互模式下使用的，你肯定也想到了。

```
>>> help(file.read)
```

用这样的方法，可以分别得到三个函数的说明：

```
read(...)
    read([size]) -> read at most size bytes, returned as a string.
    If the size argument is negative or omitted, read until EOF is
    Notice that when in non-blocking mode, less data than what was
    may be returned, even if no size parameter was given.
readline(...)
    readline([size]) -> next line from the file, as a string.
    Retain newline.  A non-negative size argument limits the maxim
    number of bytes to return (an incomplete line may be returned
    Return an empty string at EOF.
readlines(...)
    readlines([size]) -> list of strings, each a line from the fil
    Call readline() repeatedly and return a list of the lines so r
    The optional size argument, if given, is an approximate bound
    total number of bytes in the lines returned.
```

对照一下上面的说明，三者的异同就显现了。

在上面的文档中有EOF，它是什么意思？End-of-file。在维基百科中居然有对它的解释：

In computing，End Of File（commonly abbreviated EOF[1]）is a condition in a computer operating system where no more data can be read from a data source.The data source is usually called a file or stream.In general，the EOF is either determined when the reader returns null as seen in Java's BufferedReader，[2]or sometimes people will manually insert an EOF character of their choosing to signal when the file has ended.

明白EOF之后，来对比一下。

- read：如果指定了参数size，就按照该指定长度从文件中读取内容，否则，就读取全文。被读出来的内容，全部塞到一个字符串里面。这样有好处，就是东西都到内存里面了，随时取用，比较快捷；“成也萧何败也萧何”，也是因为这一点，如果文件

内容太多，内存会吃不消的。文档中已经提醒注意在“non-blocking”模式下的问题，关于这个问题，不是本节的重点，暂时不讨论。

- readline：那个可选参数size的含义同上。它以行为单位返回字符串，也就是每次读一行，依次循环，如果不限定size，直到最后一个返回的是空字符串，意味着到文件末尾了（EOF）。
- readlines：size同上。它返回的是以行为单位的列表，即相当于先执行readline()，得到每一行，然后把这一行的字符串作为列表中的元素塞到一个列表中，最后将此列表返回。

依次演示操作，即可明了。有这样一个文档，名为：you.md，其内容和基本格式如下：

```
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

分别用上述三种函数读取这个文件。

```
>>> f = open("you.md")
>>> content = f.read()
>>> content
    'You Raise Me Up\nWhen I am down and, oh my soul, so weary;\nW
>>> f.close()
```

提示：养成一个好习惯，就一定要随手关闭不用的文件。如果不关闭，它还驻留在内存中，后面又没有对它的操作，既浪费内存空间，也增加了文件安全的风险。

注意：在Python中，'\n'表示换行，这也是UNIX系统中的规范。但是，在奇葩的Windows中，用'\r\n'表示换行。不过，还好Python在处理的时候，会自动将'\r\n'转换为'\n'。

请仔细观察，得到的是一个大大的字符串，但是这个字符串里面包含着一些符号\n，因为原文中有换行符。如果用print输出这个字符串，就是这样的，其中的\n起作用了。

#建议读者耐心阅读如下内容，并到网上搜索这首歌曲听一听。

```
>>> print content
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

用readline()读取，则是这样的：

```
>>> f = open("you.md")
>>> f.readline()
'You Raise Me Up\n'
>>> f.readline()
'When I am down and, oh my soul, so weary;\n'
>>> f.readline()
'When troubles come and my heart burdened be;\n'
>>> f.close()
```

显示出一行一行读取了，每操作一次f.readline()，就读取一行，并且将指针向下移动一行，如此循环。显然，这是一种循环，或者说是可迭代的。因此，就可以用循环语句来完成对全文的读取。

```
#!/usr/bin/env python
# coding=utf-8

f = open("you.md")

while True:
    line = f.readline()
    if not line:              #到EOF，返回空字符串，则终止循环
        break
    print line ,              #注意后面的逗号

f.close()                     #别忘记关闭文件
```

将其和文件"you.md"保存在同一个目录中，我这里命名的文件名是12701.py，然后在该目录中运行python 12701.py，就看到下面的效果了：

```
~/Documents$ python 12701.py
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

也用readlines()来读取此文件：

```
>>> f = open("you.md")
>>> content = f.readlines()
>>> content
```

['You Raise Me Up\n','When I am down and,oh my soul,so weary;\n','When troubles come and my heart burdened be;\n','Then,I am still and wait here in the silence,\n','Until you come and sit awhile with me.\n','You raise me up,so I can stand on mountains;\n','You raise me up,to walk on stormy seas;\n','I am strong, when I am on your shoulders;\n','You raise me up:To more than I can be.\n']

返回的是一个列表，列表中每个元素都是一个字符串，每个字符串中的内容就是文件的一行文字，含行末的符号。显而易见，它是可以用for来循环的。

```
>>> for line in content:
...     print line ,
...
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
>>> f.close()
```

2.6.6 读很大的文件

前面已经说明了，如果文件太大，就不能用read()或者readlines()一次性将全部内容读入内存，可以使用while循环和readlin()来完成这个任务。

在Python中，一旦遇到比较特殊的、棘手的问题，往往有好的工具出现。在对付很大的文件时，就有一个模块供我们驱使：fileinput模块。

```
>>> import fileinput
>>> for line in fileinput.input("you.md"):
...     print line ,
...
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

我比较喜欢它，用起来非常得心应手，简洁明快，还有for。

对于这个模块的更多内容，读者可以自己在交互模式下利用dir()、help()去查看明白。

诚然，基本的方法也不是不能用的。

```
>>> for line in f:
...     print line ,
...
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

之所以能够如此，是因为file是可迭代的数据类型，直接用for来实现迭代过程。

2.6.7 seek()

这个函数的功能就是让指针移动。比如：

```
>>> f = open("you.md")
>>> f.readline()
'You Raise Me Up\n'
>>> f.readline()
'When I am down and, oh my soul, so weary;\n'
```

现在已经移动到第四行末尾了，看seek()的能力：

```
>>> f.seek(0)
```

意图是要回到文件的最开头，那么如果用f.readline()应该读取第一行。

```
>>> f.readline()
'You Raise Me Up\n'
```

果然如此。此时指针所在的位置还可以用tell()来显示，如：

```
>>> f.tell()
17L
>>> f.seek(4)
```

f.seek（4）就将位置定位到从开头算起的第四个字符后面，也就是"You"之后、字母"R"之前的位置。

```
>>> f.tell()
4L
```

tell()也是这么说的。这时候如果使用readline()，就是从当前位置开始到行末。

```
>>> f.readline()
```

```
'Raise Me Up\n'
>>> f.close()
```

seek()还有别的参数，具体如下：

seek（...）seek（offset[，whence]）->None.Move to new file position.

Argument offset is a byte count.Optional argument whence defaults to 0（offset from start of file，offset should be>=0）;other values are 1（move relative to current position，positive or negative），and 2（move relative to end of file，usually negative，although many platforms allow seeking beyond the end of a file）.If the file is opened in text mode，only offsets returned by tell()are legal.Use of other offsets causes undefined behavior.Note that not all file objects are seekable.

whence的值：

- 默认值是0，表示从文件开头开始计算指针偏移的量（简称偏移量）。这时offset必须是大于等于0的整数。
- 是1时，表示从当前位置开始计算偏移量。offset如果是负数，则表示从当前位置向前移动，整数表示向后移动。
- 是2时，表示相对文件末尾移动。

2.7 迭代

跟一些比较牛的程序员交流，经常听到他们嘴里会冒出不标准的英文单词，而若loop、iterate、traversal和recursion不在其内，总觉得他还不够牛。真正牛的程序员只是说“循环、迭代、遍历、递归”，然后再问“这个你懂吗？”。那么大神程序员是什么样子呢？他是扫地僧，大隐隐于市。

先搞清楚这些名词：

- 循环（loop），指的是在满足条件的情况下，重复执行同一段代码。比如，while语句。
- 迭代（iterate），指的是按照某种顺序逐个访问对象中的每一项。比如，for语句。
- 递归（recursion），指的是一个函数不断调用自身的行为。比如，以编程方式输出著名的斐波纳契数列。
- 遍历（traversal），指的是按照一定的规则访问树形结构中的每个节点，而且每个节点都只访问一次。

对于这四个听起来高深莫测的词汇，其实前面已经涉及了一个——循环（loop），本节主要介绍一下迭代（iterate），在网上搜索一下就会发现，对迭代和循环、递归进行比较的文章不少，分别从不同角度将它们进行了对比。这里暂不比较，先搞明白Python中的迭代。

当然，迭代的话题会很长，本着循序渐进的原则，这里介绍比较初级的。

2.7.1 迭代工具

要访问对象中的每个元素，可以这么做（例如一个list）：

```
>>> lst
['q', 'i', 'w', 's', 'i', 'r']
>>> for i in lst:
...     print i,
```

```
...
q i w s i r
```

除了这种方法，还可以这样：

```
>>> lst_iter = iter(lst)        #对原来的list实施了一个iter()
>>> lst_iter.next()             #要不厌其烦地一个一个手动访问
'q'
>>> lst_iter.next()
'i'
>>> lst_iter.next()
'w'
>>> lst_iter.next()
's'
>>> lst_iter.next()
'i'
>>> lst_iter.next()
'r'
>>> lst_iter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

iter()是一个内建函数。

next()就是要获得下一个元素，但是作为一名优秀的程序员，最佳品质就是“懒惰”，当然不能这样一个一个地敲，于是：

```
>>> while True:
...     print lst_iter.next()
...
Traceback (most recent call last):          #报错，而且错误跟前面一样，
  File "<stdin>", line 2, in <module>
StopIteration
```

先不管错误，再来一遍。

```
>>> lst_iter = iter(lst)          #错误暂且搁置，回头再研究
>>> while True:
...     print lst_iter.next()
...
q                                           #果然自动化地读取了
i
w
s
i
```

```
r
Traceback (most recent call last):      #读取到最后一个之后，报错，停止
  File "<stdin>", line 2, in <module>
StopIteration
```

首先了解一下上面用到的那个内置函数：iter()，官方文档中有这样一段话描述之：

iter（o[，sentinel]）

Return an iterator object.The first argument is interpreted very differently depending on the presence of the second argument.Without a second argument，o must be a collection object which supports the iteration protocol（the iter()method），or it must support the sequence protocol（the getitem()method with integer arguments starting at 0）.If it does not support either of those protocols，TypeError is raised.If the second argument，sentinel，is given，then o must be a callable object.The iterator created in this case will call o with no arguments for each call to its next()method;if the value returned is equal to sentinel，StopIteration will be raised，otherwise the value will be returned.

提炼一下此段主要的东西：

- 返回值是一个迭代器对象。
- 参数需要是一个符合迭代协议的对象或者是一个序列对象。
- next()配合与之使用。

我们常常将那些能够用诸如循环语句之类的方法来一个一个地读取元素的对象，就称为可迭代的对象。那么用来循环的（如for）就称为迭代工具。

用严格一点的语言说：所谓迭代工具，就是能够按照一定顺序扫描迭代对象的每个元素（按照从左到右的顺序）。

显然，除了for之外，还有别的迭代工具。

那么，刚才介绍的iter()的功能呢？它与next()配合使用，也有实现上述迭代工具的作用。

在Python中，甚至在其他的语言中，关于迭代的说法比较乱，主要是名词乱，刚才我们说，那些能够实现迭代的工具，称为迭代工

具，就是这些迭代工具，不少程序员都喜欢叫作迭代器。当然，这都是汉语翻译，英语就是iterator。

从例子中会发现，如果用for来迭代，当到末尾的时候就自动结束了，不会报错。如果用iter()...next()迭代，当最后一个完成之后不会自动结束，还要继续向下，但是后面没有元素了，于是就报一个称之为StopIteration的错误（这个错误的名字叫作停止迭代，这哪里是报错，分明是警告）。

读者还要关注iter()...next()迭代的一个特点。当迭代对象lst_iter被迭代结束，即每个元素都读取了一遍之后，指针就移动到了最后一个元素的后面。如果再访问，指针并没有自动返回到首位置，而是仍然停留在末位置，所以报StopIteration，想要再开始，就需要重新载入迭代对象。所以，当我在上面重新进行迭代对象赋值之后，又可以继续了。这种情况在for等类型的迭代工具中是没有的。

2.7.2 文件迭代器

有一个文件，名称是208.txt，其内容如下：

```
Learn python with qiwsir.
There is free python course.
The website is:
http://qiwsir.github.io
Its language is Chinese.
```

用迭代器来操作这个文件：

```
>>> f = open("208.txt")
>>> f.readline()                #读第一行
'Learn python with qiwsir.\n'
>>> f.readline()                #读第二行
'There is free python course.\n'
>>> f.readline()                #读第三行
'The website is:\n'
>>> f.readline()                #读第四行
'http://qiwsir.github.io\n'
>>> f.readline()                #读第五行，也就是最后一行
'Its language is Chinese.\n'
>>> f.readline()                #无内容了，但是不报错，返回空
''
```

以上演示的是用readline()一行一行地读。当然，在实际操作中，我们是绝对不能这样做的，一定要让它自动进行，比较常用的方法是：

```
>>> for line in f:
...     print line,
```

这段代码之所没有打印出东西来，是因为经过前面的迭代，指针已经移到了最后。这就是迭代的一个特点，要小心指针的位置。

```
>>> f = open("208.txt")      #从头再来
>>> for line in f:
...     print line,
...
Learn python with qiwsir.
There is free python course.
The website is:
http://qiwsir.github.io
Its language is Chinese.
```

这种方法是读取文件常用的。另外一个readlines()也可以。但是，需要小心操作。

上面过程用next()也能够实现。

```
>>> f = open("208.txt")
>>> f.next()
'Learn python with qiwsir.\n'
>>> f.next()
'There is free python course.\n'
>>> f.next()
'The website is:\n'
>>> f.next()
'http://qiwsir.github.io\n'
>>> f.next()
'Its language is Chinese.\n'
>>> f.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

如果用next()，就可以直接读取每行的内容，这说明文件是天然的可迭代对象，不需要用iter()转换。

再有，我们用for来实现迭代，在本质上就是自动调用next()，只不过这个工作已经让for偷偷地替我们干了，到这里应该给for取另外一个名字：雷锋。

列表解析也能够作为迭代工具，在研究列表的时候，想必已经清楚了。那么对文件，是否可以用？试一试：

```
>>> [ line for line in open('208.txt') ]
    ['Learn python with qiwsir.\n', 'There is free python course.\
```

至此，看官难道还不为列表解析的强大魅力所折服吗？真的很强大。

其实，迭代器远远不止上述这么简单，下面我们随便列举一些，在Python中还可以这样得到迭代对象中的元素。

```
>>> list(open('208.txt'))
    ['Learn python with qiwsir.\n', 'There is free python course.\
>>> tuple(open('208.txt'))
    ('Learn python with qiwsir.\n', 'There is free python course.\
>>> "$$$".join(open('208.txt'))
    'Learn python with qiwsir.\n$$$There is free python course.\n$
>>> a,b,c,d,e = open("208.txt")
>>> a
'Learn python with qiwsir.\n'
>>> b
'There is free python course.\n'
>>> c
'The website is:\n'
>>> d
'http://qiwsir.github.io\n'
>>> e
'Its language is Chinese.\n'
```

上述方式在编程实践中不一定用得上，只是秀一下可以这么做，但不是非要这么做。

字典是否可迭代？可以。读者不妨自己仿照前面的方法摸索一下（其实前面已经用for迭代过了，这次请用iter()...next()手动一步一步迭代）。

第3章　函数

对于人类来讲，函数能够发展到这个数学思维层次是一个飞跃。可以说，函数的提出直接加快了现代科技和社会的发展，现代的任何科技门类，乃至经济学、政治学、社会学等，都已经普遍使用函数。

下面一段是来自维基百科关于函数的词条。

函数这个数学名词是莱布尼兹在1694年开始使用的，以描述曲线的一个相关量，如曲线的斜率或者曲线上的某一点。莱布尼兹所指的函数现在被称作可导函数，数学家之外的普通人一般接触到的函数即属此类。对于可导函数可以讨论它的极限和导数。此两者描述了函数输出值的变化同输入值变化的关系，是微积分学的基础。

中文的“函数”一词由清朝数学家李善兰译出。其《代数学》书中解释：“凡此变量中函（包含）彼变量者，则此为彼之函数”。

函数，从简单到复杂，各式各样。但不管什么样子的函数，都可以用图3-1概括。

下面说明Python中的函数和相关问题。

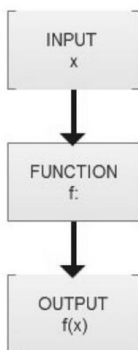


图3-1　函数

3.1 理解函数

在中学数学中，可以用这样的方式定义函数：$y=4x+3$，这就是一个一次函数，当然，也可以写成：$f(x)=4x+3$。其中x是变量，它可以代表任何数。

当$x=2$时，代入到上面的函数表达式：

$f(2)=4*2+3=11$

所以：$f(2)=11$

但是，这并不是函数的全部，其实在函数中，并没有规定变量只能是一个数，它可以是馒头、还可以是苹果，不知道读者是否对函数有这个层次的理解，继续阅读会理解更深刻。

3.1.1 变量不仅仅是数

变量x只能是任意数吗？其实，一个函数，就是一个对应关系。读者尝试着将上面表达式的x理解为馅饼，$4x+3$，就是4个馅饼加上3（一般来讲，单位是统一的，但你非让它不统一也无妨），这个结果对应着另外一个东西，那个东西比如说是iPhone。或者说可以理解为4个馅饼加3就对应一个iPhone，这就是所谓的映射关系。

所以，x不仅仅是数，还可以是你认为的任何东西。

变量本质上是占位符

函数中为什么变量用x？这是一个有趣的问题，自己搜索一下，看能不能找到答案。

我也不清楚原因。不过，我清楚地知道，变量可以用x，也可以用别的符号，甚至用alpha、beta这样的字母组合也可以。

变量在本质上就是一个占位符，这是一针见血的理解。什么是占位符？就是先把那个位置用变量占上，表示这里有一个东西，至于这

个位置放什么东西，以后再说，反正先用一个符号占着这个位置（占位符）。

其实在高级语言编程中，变量比我们在初中数学中学习的要复杂。但是，先不管那些，现在，就按照初中数学的难度来研究Python中的变量。

在Python中，通常用小写字母来命名变量，也可以在其中加上下画线，以表示区别。

3.1.2　建立简单函数

```
>>> a = 2
>>> y=3*a+2
>>> y
8
```

这种方式建立的函数，与在初中数学中学习的没有什么区别。当然，这种方式的函数在编程实践中没什么用途，仅仅在这里冒出来，后面绝对不用这个形式了。

输入a＝3，然后输入y，看看得到什么结果呢？

```
>>> a=2
>>> y=3*a+2
>>> y
8
>>> a=3
>>> y
8
```

是不是很奇怪？已经让a等于3了，为什么结果y还是8？

还记得前面已经学习过的关于“变量赋值”的原理吗？a＝2的含义是将2这个对象贴上了变量a的标签，经过计算，得到了8，之后变量y引用了对象8。当变量a引用的对象修改为3的时候，但是y引用的对象还没有变，所以还是8。再计算一次，y的连接对象就变了：

```
>>> a=3
>>> y
8
```

```
>>> y=3*a+2
>>> y
11
```

特别注意，如果没有预先令a=2，直接写函数表达式则会报错。

```
>>> y=3*a+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

注意看错误提示，a是一个变量，提示中告诉我们这个变量没有定义。显然，如果函数中要使用某个变量，不得不提前定义出来，定义方法就是给这个变量赋值。

3.1.3 建立实用的函数

上面用命令方式建立函数还不够“正规化”，那么就来写一个.py文件吧。

例如，下面的代码：

```
#!/usr/bin/env python
#coding:utf-8

def add_function(a, b):
    c = a + b
    print c

if __name__ == "__main__":
    add_function(2, 3)
```

然后将文件保存，我把它命名为20101.py，你根据自己的喜好取个名字。

然后就进入到那个文件夹，运行这个文件，出现下面的结果：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105-1.py  105.py  106-1.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python 106-1.py
5
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ |
```

你运行的结果是什么？如果没有得到上面的结果，就要非常认真

地检查代码，注意，冒号和空格都得一样，因为冒号和空格也很重要。

下面开始庖丁解牛。

- def add_function（a，b）：这是函数的开始。在声明要建立一个函数的时候，一定要使用def（def就是英文define的前三个字母），意思就是告知计算机，这里要声明一个函数；add_function是这个函数名称，取名字是有讲究的，就好比你的名字一样。在Python中取名字的讲究就是要有一定意义，能够从名字中看出这个函数是用来干什么的。从add_function这个名字中，可以看出它是用来计算加法的（严格地说是把两个对象“相加”，这里相加的含义是比较宽泛的，包括对字符串等相加）。（a，b）这个括号里面的是这个函数的参数，也就是函数变量。冒号非常非常重要，如果少了，就会报错。冒号的意思就是下面开始真正的函数内容了。
- c＝a＋b：这一行比上一行要缩进四个空格。这是Python的规定，要牢记，不可丢掉，丢了就报错。然后这句话就是将两个参数相加，结果赋值与另外一个变量c。
- print c：还是提醒注意，缩进四个空格。将得到的结果c的值打印出来。
- if_name_＝＝"_main_"：这句话先照抄，不解释（后面会做解释的）。注意的地方就是不缩进了。
- add_function（2，3）：这才是真正调用前面建立的函数，并且传入两个参数：a＝2，b＝3。仔细观察传入参数的方法，就是把2放在a那个位置，3放在b那个位置。

解牛完毕，做个总结。

定义函数的格式为：

```
def 函数名(参数1，参数2，...，参数n)：
    函数体（语句块）
```

是不是样式很简单呢？

几点说明：

- 函数名的命名规则要符合Python中的命名要求。一般用小写字母和单下画线、数字等组合。
- def是定义函数的关键词，这个简写来自英文单词define。

- 函数名后面是圆括号，括号里面，可以有参数列表，也可以没有参数。
- 千万不要忘记了括号后面的冒号。
- 函数体（语句块），相对于def缩进，按照Python的习惯，缩进四个空格。

再通过简单的例子，深入理解上面的要点：

```
>>> def name():            #定义一个无参数的函数
...     print "qiwsir"         #缩进4个空格
...
>>> name()                 #调用函数，打印结果
qiwsir

>>> def add(x, y):         #定义一个非常简单的函数
...     return x+y            #缩进4个空格
...
>>> add(2, 3)              #计算2+3
5
```

注意上面的add（x，y）函数，没有特别规定参数“x，y”的类型。其实，这句话本身就是错的，前面已经多次提到，在Python中，变量无类型，只有对象才有类型，这句话应该说成：“x，y”并没有严格规定其所引用的对象类型。这是Python跟某些语言很大的区别，在有些语言中，需要在定义函数的时候告诉函数参数的数据类型，Python不用那样做。

为什么？读者不要忘记了，这里的所谓参数跟前面说的变量本质上是一回事。只有当用到该变量的时候，才建立变量与对象的对应关系，否则，关系不建立。只有对象才有类型，那么，在add（x，y）函数中，“x，y”在引用对象之前是完全飘忽的，没有被贴在任何一个对象上，换句话说它们有可能引用任何对象，只要后面的运算许可。如果后面的运算不许可，则会报错。

```
>>> add("qiw","sir")
'qiwsir'

>>> add("qiwsir",4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add
TypeError: cannot concatenate 'str' and 'int' objects       #仔细阅
```

从实验结果中发现：x＋y的意义完全取决于对象的类型。在Python中，将这种依赖关系称之为多态。对于Python中的多态问题以后还会遇到，这里仅仅以此例子显示一番。请读者要留心注意：Python中为对象编写接口，而不是为数据类型编写。读者先留心一下这句话，或者记住它，随着学习的深入，会领悟到其真谛的。

此外，也可以将函数通过赋值语句与某个变量建立引用关系：

```
>>> result = add(3, 4)
>>> result
7
```

这其实解释了函数的一个秘密：add（x，y）被运行之前，在计算机内是不存在的，直到代码运行到这里的时候，在计算机中就建立起来了一个对象，这就如同前面所学习过的字符串、列表等类型的对象一样，运行add（x，y）之后，也建立了一个add（x，y）的对象，这个对象与变量result可以建立引用关系，并且add（x，y）将运算结果返回。请注意函数中的return，它的作用就是要把函数的结果返回，从而得到这个函数的返回值。于是，通过result就可以查看运算结果。

如果对上面一段感觉有点吃力或者晕乎，那就再读一遍。若实在搞不明白，也别担心，随着学习的深入，会明白的。

3.1.4　关于命名

到现在为止，我们已经接触过变量的命名和函数的命名问题，似乎已经到了将命名问题进行总结的时候了。

所谓“名不正言不顺”，取名字或者给什么东西命名，常常是天大的事情。

Python也很在乎名字问题，其实，所有高级语言对名字都有要求。为什么呢？因为如果命名乱了，计算机就有点不知所措了。看Python对命名的一般要求。

- 文件名：全小写，可使用下画线。
- 函数名：小写，可以用下画线风格单词以增加可读性。如：my_function、my_example_function。注意：混合大小写仅被

允许用于这种风格已经占据优势的时候，以便保持向后兼容。有的人喜欢用这样的命名风格：myFunction，除了第一个单词首字母外，后面的单词首字母大写。这也是可以的，因为在某些语言中就习惯如此。
- 函数的参数：如果一个函数的参数名称和保留的关键字冲突，通常使用一个后缀下画线。
- 变量：变量名全部小写，由下画线连接各个单词。如color＝WHITE，this_is_a_variable＝1。

其实，关于命名的问题，还有不少争论，最典型的是所谓匈牙利命名法、驼峰命名等。

Python本身有一套命名的官方规范，读者可以参考：http://legacy.python.org/dev/peps/pep-0008/#prescriptive-naming-conventions。

3.1.5 调用函数

前面的例子中已经有了一些关于调用的问题，为了深入理解，把这个问题单独拿出来看看。

为什么要写函数？从理论上说，不用函数也能够编程，我们在前面已经写了程序，就没有写函数，当然，用Python的内建函数姑且不算。之所以使用函数主要原因是：

- 降低编程的难度，通常将一个复杂的大问题分解成一系列更简单的小问题，然后将小问题继续划分成更小的问题，当问题细化为足够简单时，就可以分而治之。为了实现这种分而治之的设想，就要通过编写函数，将各个小问题逐个击破，再集合起来，解决大的问题。（请注意，分而治之的思想是编程的一个重要思想，所谓“分治”方法也。）
- 代码重用。在编程的过程中，比较忌讳同样一段代码不断重复，所以，定义一个函数，可以多次被使用。当然，后面我们还会讲到“模块”，还可以把函数放到一个模块中供其他程序员使用。也可以使用其他程序员定义的函数（比如import，前面已经用到了，就是应用了别人——创造Python的人——写好的函数），这就避免了重复劳动，提高了工作效率。

这样看来，使用函数还是很有必要的。下面就看函数怎么调用

吧。以add（x，y）为例，前面已经演示了基本调用方式，此外，还可以这样：

```
>>> def add(x, y):
...     print "x=", x
...     print "y=", y
...     return x + y
...
>>> add(10, 3)           #x=10,y=3
x= 10
y= 3
13

>>> add(3, 10)          #x=3,y=10
x= 3
y= 10
13
```

所谓调用，最关键是要弄清楚如何给函数的参数赋值。这里就是按照参数次序赋值，根据参数的位置，值与之对应。

还可以直接把赋值语句写到里面，就明确了参数和对象的关系。

```
>>> add(x=10, y=3)       #同上
x= 10
y= 3
13
```

当然，这时候顺序就不重要了，也可以这样：

```
>>> add(y=10, x=3)
x= 3
y= 10
13
```

在定义函数的时候，参数可以像前面那样等待被赋值，也可以定义的时候就赋给一个默认值。例如：

```
>>> def times(x, y=2):
...     print "x=", x
...     print "y=", y
...     return x * y
...
>>> times(3)
x= 3
```

```
y= 2
6

>>> times(x=3)
x= 3
y= 2
6
```

如果不给那个有默认值的参数传递值（赋值的另外一种说法），那么它就是用默认的值。如果给它传一个，则采用新赋给它的值。如下：

```
>>> times(3, 4)              #x=3,y=4,y的值不再是2
x= 3
y= 4
12

>>> times("qiwsir")         #再次体现了多态特点
x= qiwsir
y= 2
'qiwsirqiwsir'
```

给各位读者提一个思考题，请在闲暇之余用Python完成：写两个数的加、减、乘、除的函数，然后用这些函数，完成简单的计算。

3.1.6 注意事项

把编写程序的注意事项单独作为一个小节提出，目的是提醒读者注意，因为这些方面的问题，往往成为困扰读者（特别是初学者）的麻烦。

- 别忘了冒号。一定要记住符合语句首行末尾输入“：”（if，while，for等的第一行）。
- 从第一行开始。要确定顶层（无嵌套）程序代码从第一行开始。
- 空白行在交互模式提示符下很重要。模块文件中符合语句内的空白行常被忽视。但是，当你在交互模式提示符下输入代码时，空白行则会结束语句。
- 缩进要一致。避免在块缩进中混合制表符和空格。
- 使用简洁的for循环，而不是while or range。相比，for循环更易写，运行起来也更快。

- 要注意赋值语句中的可变对象。
- 不要期待在原处修改的函数会返回结果，比如list.append()，这在可修改的对象中要特别注意。
- 调用函数时，函数名后面一定要跟随着括号，有时候括号里面就是空空的，有时候里面放参数。
- 不要在导入和重载（先记住有这样一个名词，后面会反复出现，慢慢就能理解含义了，实在忍不住就搜索一下。）中使用扩展名或路径。

以上各点如果有不理解的也不要紧，在以后编程中，时不时地回来复习一下，能不断领悟其内涵。

3.1.7　返回值

前面写的函数有点缺憾，不知道读者是否觉察到了，即结果是用print语句打印出来的。这是实际编程中广泛使用的吗？肯定不是。因为函数在编程中是一段具有抽象价值的代码，一般情况下，用它得到一个结果，这个结果要用在其他的运算中。所以，不能仅仅局限在把某个结果打印出来，函数必须返回一个结果。

为了能够说明清楚，先编写一个函数。

根据高德纳（Donald Ervin Knuth）的《计算器程序设计艺术》（The Art of Computer Programming），1150年印度数学家Gopala和金月在研究箱子包装对象长宽刚好为1和2的可行方法数目时，首先描述这个数列。在西方，最先研究这个数列的人是比萨的李奥纳多（意大利人斐波那契Leonardo Fibonacci），他在描述兔子生长的数目时用上了这个数列。

第一个月初有一对刚诞生的兔子；第二个月之后（第三个月初）它们可以生育，每月每对可生育的兔子会诞生下一对新兔子；兔子永不死去。

假设在n月有可生育的兔子总共a对，n＋1月就总共有b对。在n＋2月必定总共有a＋b对：＞因为在n＋2月的时候，前一月（n＋1月）的b对兔子可以存留至第n＋2月（在当月属于新诞生的兔子尚不能生育）。而新生育出的兔子对数等于所有在n月就已存在的a对。

下面定义一个能够得到斐波那契数列的函数，从而可以实现任意

的数列。

参考代码：

```
#!/usr/bin/env python
# coding=utf-8

def fibs(n):
    result = [0, 1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result

if __name__ == "__main__":
    lst = fibs(10)
    print lst
```

把含有这些代码的文件保存成名为20202.py的文件。

在这个文件中，首先定义了一个函数，名字叫作fibs，其参数是输入一个整数（当然，函数并没有对此进行限制，也没有必要限制，这就是Python的脾气和性格。）。然后，通过lst＝fibs（10）调用这个函数。这里参数给的是10，就意味着要得到n＝10的斐波那契数列。

运行后打印数列：

```
$ python 20202.py
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

当然，如果要换n的值，只需要在调用函数的时候，修改一下参数即可，这才体现出函数的优势呢。

观察fibs函数，最后有一个语句return result，意思是将变量result的值返回。返回给谁呢？一般这类函数调用的时候，要通过类似lst＝fibs（10）的语句，那么返回的那个值就被变量lst贴上了，通过lst就能得到该值。如果没有这个赋值语句，虽然函数照样返回值，但是它飘忽在内存中，我们无法抓出来，并且最终还被当作垃圾被Python回收了。

上面的函数只返回了一个返回值（是一个列表），若你需要返回多个也是可以的，只不过是以元组形式返回。

```
>>> def my_fun():
...     return 1,2,3
...
>>> a = my_fun()
>>> a
(1, 2, 3)
```

有的函数没有renturn一样执行完毕，就算也干了某些活儿吧。事实上，不是没有返回值，只不过是None。比如这样一个函数：

```
>>> def my_fun():
...     print "I am doing somthing."
```

在交互模式下构造一个很简单的函数，注意，这是构造了一个简单函数，如果是复杂的函数，千万不要在交互模式下做。如果你非要做，则会尝到苦头。

这个函数的作用就是打印出一段话，即执行这个函数就能打印出那段话，但是没有return。

```
>>> a = my_fun()
I am doing somthing.
```

我们再看看那个变量a，到底是什么。

```
>>> print a
None
```

这就是只干活儿没有return的函数，返回给变量的是一个None。这种模样的函数通常不用上述方式调用，而采用下面的方式，因为他们返回的是None，似乎这个返回值的利用价值不高，所以不用找一个变量来接受返回值了。

```
>>> my_fun()
I am doing somthing.
```

特别注意那个return，它还有一个作用。观察下面的函数和执行结果，看看能不能发现它的另外一个作用。

```
>>> def my_fun():
...     print "I am coding."
...     return
```

```
...     print "I finished."
...
>>> my_fun()
I am coding.
```

看出玄机了吗？在函数中，本来有两个print语句，但是中间插入了一个return，仅仅是一个return。当执行函数的时候，只执行了第一个print语句，第二个并没有执行。这是因为执行第一个语句之后，遇到了return，它告诉函数要返回，即中断函数体内的流程，离开这个函数，结果第二个print就没有被执行。所以，return在这里就有了一个作用，结束正在执行的函数。有点类似循环中的break的作用，仅仅是类似罢了。

3.1.8 函数中的文档

“程序在大多数情况下是给人看的，只是偶尔被机器执行一下。”

所以，写程序必须要写注释。有人宣称，好代码是不需要注释的。对此，我表示赞同，但是前提是“好代码”。如果写不出“好代码”，是不是就需要注释了呢？大多数程序员，包括我在内，都不能保证自己的代码属于人见人爱的“好代码”，所以，注释是必要的。

前面已经有过说明，如果某一行用#开始，Python就不执行它后面的内容（Python看不到它，但是人能看到），则这一行就作为注释存在。

除了这样的一句之外，一般在每个函数名字的下面，还要写一写文档，以此来说明这个函数的用途。

```
#!/usr/bin/env python
# coding=utf-8

def fibs(n):
    """
    This is a Fibonacci sequence.
    """
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result

if __name__ == "__main__":
```

```
lst = fibs(10)
print lst
```

在这个函数的名称下面，用三个引号的方式，包裹着对这个函数的说明，即函数文档。

为了能清晰地了解函数文档，先写一个简单的函数例子。

```
>>> def add(x, y):
        return x + y
>>> dir(add)
['__call__', '__class__', '__closure__', '__code__', '__defaults__
```

请读者用你那一双慧眼找一找，有没有这样一个东西：_doc_，它里面就记录着这个函数的文档信息。

```
>>> print add.__doc__
None
```

居然是None，这没有错。因为上面的函数根本就没有写文档内容，如果不是None那才是有问题呢。

下面增加点儿东西，就是函数文档。

```
>>> def add(x, y):
        """ add x and y."""
        return x + y

>>> print add.__doc__
add x and y.
```

这显示了函数文档的内容。这样为每个函数都做了一个简要说明，让调用这个函数的人明白函数的作用和使用意图，我代表另外一个你不认识的程序员向勤劳写文档的你表示感谢，因为你的勤劳免去了我们的烦恼。

3.2 名词辨析

在函数中，有一些名词容易让人糊涂，甚至常常谈起，但不知道其含义，需要对这些名词深究一番。

3.2.1 参数和变量

参数，是貌似比较复杂的，所以要深入讨论。

在程序员嘴里，你或许听说过“形参”、“实参”、“参数”等名词，到底指什么呢？

在定义函数的时候（def来定义函数，称为def语句），函数名后面的括号里如果有变量，它们通常被称为“形参”。调用函数的时候，给函数提供的值叫作“实参”，或者“参数”。

其实，如果你区别不开，也不会耽误你写代码，这只不过类似孔乙己先生知道茴香豆的茴字有多少种写法罢了。但是，我居然碰到过某公司面试官问这种问题。

我们就简化一下，笼统地把函数括号里面的变量叫作参数吧，当然，你叫作变量也无妨，只要大家都知道指的是什么东西就好了。虽然这样说会引起某些认真的人来喷口水，但不用担心，反正本书已经声明是很“水”的了。

但如果有人较真，非要让你区分，为了显示你的水平，你可以引用微软网站上的说明。我认为这段说明高度抽象，而且意义涵盖深远。摘抄过来，请读一读，看看是否理解。

参数和变量之间的差异（Visual Basic）

多数情况下，过程必须包含有关调用环境的一些信息。执行重复或共享任务的过程对每次调用使用不同的信息。此信息包含每次调用过程时传递给它的变量、常量和表达式。

若要将此信息传递给过程，过程先要定义一个形参，然后调用代

码将一个实参传递给所定义的形参。您可以将形参当作一个停车位，而将实参当作一辆汽车。就像一个停车位可以在不同时间停放不同的汽车一样，调用代码在每次调用过程时可以将不同的实参传递给同一个形参。

形参表示一个值，过程希望您在调用它时传递该值。

当您定义Function或Sub过程时，需要在紧跟过程名称的括号内指定形参列表。对于每个形参，您可以指定名称、数据类型和传入机制（ByVal（Visual Basic）或ByRef（Visual Basic））。您还可以指示某个形参是可选的。这意味着调用代码不必传递它的值。

每个形参的名称均可作为过程内的局部变量。形参名称的使用方法与其他任何变量的使用方法相同。

实参表示在您调用过程时传递给过程形参的值。调用代码在调用过程时提供参数。

调用Function或Sub过程时，需要在紧跟过程名称的括号内包括实参列表。每个实参均与此列表中位于相同位置的那个形参相对应。

与形参定义不同，实参没有名称。每个实参就是一个表达式，它包含零或多个变量、常数和文本。求值的表达式的数据类型通常应与为相应形参定义的数据类型相匹配，并且在任何情况下，该表达式值都必须可转换为此形参类型。

看完这段引文会发现里面有几个关键词：参数、变量、形参、实参。本来想弄清楚参数和变量，结果又冒出另外两个词，更混乱了。请少安毋躁，在编程业界，类似的东西有很多名词。下次听到有人说这些就不用害怕啦，反正自己听过了。

在Python中，没有这么复杂。

看完上面让人晕头转向的引文之后，再看下面的代码就会豁然开朗了。

```
>>> def add(x):       #x是参数，准确说是形参
...     a = 10                  #a是变量
...     return a + x            #x就是那个形参作为变量，其本质是要传递赋给
...
>>> x = 3             #x是变量，只不过在函数之外
>>> add(x)            #这里的x是参数，但是它由前面的变量x传递对象3
13
```

```
>>> add(3)              #把上面的过程合并了
13
```

至此，读者是否清楚了一点点。其实没有那么复杂。关键要理解函数名括号后面的东西的作用是传递值。

3.2.2 全局变量和局部变量

虽然是讲参数，但是关于全局变量和局部变量的区别也要先弄清楚，因为关系到函数内外有别的大事。

下面这段代码中有一个函数funcx()，这个函数里面有一个变量x＝9，在函数的前面也有一个变量x＝2。

```
x = 2

def funcx():
    x = 9
    print "this x is in the funcx:-->", x

funcx()
print "--------------------------"
print "this x is out of funcx:-->", x
那么，这段代码输出的结果是什么呢？看：
this x is in the funcx:--> 9
--------------------------
this x is out of funcx:--> 2
```

从输出看出，运行funcx()，输出了funcx()里面的变量x＝9；然后执行代码中的最后一行，print"this x is out of funcx:-->"，x

要特别关注的是，前一个x输出的是函数内部的变量x；后一个x输出的是函数外面的变量x。两个变量彼此没有互相影响，虽然都是x。从这里看出，两个x各自在各自的领域内起到作用。

把那个只在函数体内（某个范围内）起作用的变量称之为局部变量。

有局部就有对应的全部，感觉很别扭，局部对应的是全局，不是全部，所以又来了一个名词：全局变量。

```
x = 2
```

```
def funcx():
    global x                           #跟上面函数的不同之处
    x = 9
    print "this x is in the funcx:-->", x

funcx()
print "--------------------------"
print "this x is out of funcx:-->",x
```

以上两段代码的不同之处在于，后者在函数内多了一个global x，这句话的意思是在声明x是全局变量，也就是说这个x跟函数外面的那个x是同一个，接下来通过x＝9将x的引用对象变成了9。所以，就出现了下面的结果。

```
this x is in the funcx:--> 9
--------------------------
this x is out of funcx:--> 9
```

好似全局变量能力很强悍，能够统帅函数内外。但是，要注意，这个东西要慎重使用，因为往往容易带来变量的混乱。内外有别，在程序中一定要注意。

3.2.3 命名空间

命名空间是一个比较不容易理解的概念，特别是对于初学者而言，似乎它很缥缈。我在维基百科中看到对它的定义，不仅定义比较好，连里面的例子都不错。所以，抄录下来，帮助读者理解这个名词。

命名空间（英语：Namespace）表示标识符（identifier）的可见范围。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。在一个新的命名空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其他命名空间中。

例如，设Bill是X公司的员工，工号为123，而John是Y公司的员工，工号也是123。由于两人在不同的公司工作，可以使用相同的工号来标识而不会造成混乱，这里每个公司就表示一个独立的命名空间。如果两人在同一家公司工作，其工号就不能相同了，否则在支付工资时便会发生混乱。

这一特点是使用命名空间的主要理由。在大型的计算机程序或文档中，往往会出现数百或数千个标识符。命名空间提供一隐藏区域标识符的机制。通过将逻辑上相关的标识符组织成相应的命名空间，可使整个系统更加模块化。

在编程语言中，命名空间是对作用域的一种特殊的抽象，它包含了处于该作用域内的标识符，且本身也用一个标识符来表示，这样便将一系列在逻辑上相关的标识符用一个标识符组织了起来。许多现代编程语言都支持命名空间。在一些编程语言（例如C＋＋和Python）中，命名空间本身的标识符也属于一个外层的命名空间，即命名空间可以嵌套，构成一个命名空间树，树根则是无名的全局命名空间。

函数和类的作用域可被视作隐式命名空间，它们和可见性、可访问性和对象生命周期不可分割地联系在一起。

显然，用“命名空间”或者“作用域”这样的名词，就是因为有了函数（后面还会有类）之后，在函数内外都可能有外形一样的符号（标识符），在Python中（乃至于其他高级语言），为了区分此变量非彼变量（虽然外形一样），需要用这样的东西来框定每个变量所对应的值（发生作用的范围）。

前面已经讲过，变量和对象（就是所变量所对应的值）之间的关系是：变量类似标签，贴在了对象上，即通过赋值语句实现了一个变量标签对应一个数据对象（值），这种对应关系让你想起了什么？映射！Python中唯一的映射就是字典，里面有“键/值对”。变量和值的关系就有点像“键”和“值”的关系。有一个内建函数vars，可以帮助我们研究一下这种对应关系。

```
>>> x = 7
>>> scope = vars()
>>> scope['x']
7
>>> scope['x'] += 1
>>> x
8
>>> scope['x']
8
```

既然如此，诚如前面的全局变量和局部变量，即使是同样一个变量名称，但是它在不同范围（用“命名空间”更专业）对应不同的值。

3.3 参数收集

在函数中，参数的个数有时候是一个，比如一个用来计算圆面积的函数，它所需要的参数就是半径（πr^2），这个函数的参数是确定的。

然而，这个世界不总是这么简单的，有时候参数个数是多个，甚至不是确定的，不确定性或许更是常态。如果读者了解量子力学就更理解真正的不确定性了。当然，不用研究量子力学也一样能够体会到，世界充满里了不确定性。也就是说，我们还要解决函数的参数个数不确定的情况。

3.3.1 参数收集

Python用这样的方式解决参数个数的不确定性：

```
def func(x,*arg):
    print x
    result = x
    print arg
    for i in arg:
        result +=i
    return result
print func(1, 2, 3, 4, 5, 6, 7, 8, 9)              #赋给函数的参数个数不仅
```

运行此代码后，得到如下结果：

```
1                        #这是第一个print，参数x得到的值是1
(2, 3, 4, 5, 6, 7, 8, 9) #这是第二个print，参数arg得到的是一个元组
45                       #最后的计算结果
```

从上面例子可以看出，如果输入的参数个数不确定，其他参数全部通过*arg，以元组的形式由arg收集起来。对照上面的例子不难发现：

- 值1传给了参数x。
- 值2、3、4、5、6、7、8、9被塞入一个tuple里面，传给了

arg。

为了能够更明显地看出arg（名称可以不一样，但是符号*必须要有），可以用下面的一个简单函数来演示：

```
>>> def foo(*args):
...     print args          #打印通过这个参数得到的对象
```

下面演示分别传入不同的值，通过参数*args得到的结果：

```
>>> foo(1, 2, 3)
(1, 2, 3)

>>> foo("qiwsir", "qiwsir.github.io", "python")
('qiwsir', 'qiwsir.github.io', 'python')

>>> foo("qiwsir", 307, ["qiwsir",2], {"name":"qiwsir", "lang":"pyt
('qiwsir', 307, ['qiwsir', 2], {'lang': 'python', 'name': 'qiwsir'
```

不管是什么，都一股脑地塞进了tuple中。

```
>>> foo("python")
('python',)
```

即使只有一个值，也是用tuple收集它。特别注意，在tuple中，如果只有一个元素，后面要有一个逗号。

还有一种可能，就是不给那个*args传值，这也是许可的。例如：

```
>>> def foo(x, *args):
...     print "x:", x
...     print "tuple:", args
...
>>> foo(7)
x: 7
tuple: ()
```

这时候*args收集到的是一个空的tuple。

在各类编程语言中，常常会遇到以foo、bar、foobar等之类的命名，不管是对变量、函数还是后面要讲到的类。这是什么意思呢？下面是来自维基百科的解释。

在计算机程序设计与计算机技术的相关文档中，术语foobar是一个常见的无名氏化名，常被作为“伪变量”使用。

从技术上讲，“foobar”很可能在20世纪60年代至70年代初通过迪吉多的系统手册传播开来。另一种说法是，“foobar”可能来源于电子学中反转的foo信号；这是因为如果一个数字信号是低电平有效（即负压或零电压代表“1”），那么在信号标记上方一般会标有一根水平横线，而横线的英文即为“bar”。在《新黑客辞典》中，还提到“foo”可能早于“FUBAR”出现。

单词“foobar”或分离的“foo”与“bar”常出现于程序设计的案例中，如同Hello World程序一样，它们常被用于向学习者介绍某种程序语言。“foo”常被作为函数／方法的名称，而“bar”则常被用作变量名。

除了用args这种形式的参数接收多个值之外，还可以用“**kargs”的形式接收数值，不过这次有点不一样：

```
>>> def foo(**kargs):
...     print kargs
...
>>> foo(a=1, b=2, c=3)     #注意观察这次赋值的方式和打印的结果
{'a': 1, 'c': 3, 'b': 2}
```

如果这次还用foo（1，2，3）的方式，会有什么结果呢？

```
>>> foo(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 0 arguments (3 given)
```

如果用**kargs的形式收集值，会得到字典类型的数据，但是，需要在传值的时候说明“键”和“值”，因为在字典中是以“键值”对形式出现的。

把上面的几种情况综合起来，看看有什么效果。

```
>>> def foo(x, y, z, *args, **kargs):
...     print x
...     print y
...     print z
...     print args
...     print kargs
```

```
...
>>> foo('qiwsir', 2, "python")
qiwsir
2
python
()
{}
>>> foo(1, 2, 3, 4, 5)
1
2
3
(4, 5)
{}
>>> foo(1, 2, 3, 4, 5, name="qiwsir")
1
2
3
(4, 5)
{'name': 'qiwsir'}
```

这样就能够足以应付各种各样的参数要求了。

3.3.2 更优雅的方式

```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
```

这是通常的函数调用方法，在前面已经屡次用到。这种方法简单明快，很容易理解。但是，世界总是多样性的，甚至在某种情况下你秀出下面的方式可能更优雅。

```
>>> bars = (2, 3)
>>> add(*bars)
5
```

先把要传的值放到元组中，赋值给一个变量bars，然后用add（*bars）的方式，把值传到函数内，这有点像前面收集参数的逆过程。注意，元组中元素的个数要跟函数所要求的变量个数一致。如果这样：

```
>>> bars = (2, 3, 4)
>>> add(*bars)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes exactly 2 arguments (3 given)
```

就报错了。

这是使用一个星号*，以元组形式传值，如果用**的方式，是不是应该以字典的形式传值呢？理当如此。

```
>>> def book(author, name):
...     print "%s is writing %s" % (author, name)
...
>>> bars = {"name":"Starter learning Python", "author":"Kivi"}
>>> book(**bars)
Kivi is writing Starter learning Python
```

这种调用函数传值的方式很少用到，至少在我的编程实践中用得不多，但不代表读者不用，这或许是习惯问题。

3.3.3 综合贯通

Python中函数的参数通过赋值的方式来传递引用对象。下面总结常见的函数参数定义方式，来理解参数传递的流程。

```
def foo(p1, p2, p3,...)
```

这种方式最常见了，列出有限个数的参数，并且彼此之间用逗号隔开。在调用函数的时候，按照顺序对参数进行赋值，特别要注意的是，参数的名字不重要，重要的是位置。而且，必须数量一致，一一对应。第一个对象（可能是数值、字符串等）对应第一个参数，第二个对象对应第二个参数，如此对应，不得偏左也不得偏右。

```
>>> def foo(p1, p2, p3):
...     print "p1==>",p1
...     print "p2==>",p2
...     print "p3==>",p3
...
>>> foo("python", 1, ["qiwsir","github","io"])
p1==> python
```

```
p2==> 1
p3==> ['qiwsir', 'github', 'io']

>>> foo("python")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 3 arguments (1 given)     #注意看报错信
```

```
>>> foo("python", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 3 arguments (4 given)
```

def foo(p1=value1, p2=value2,...)

这种方式要求把参数和值都写上，貌似这样就不乱了，很明确呀，颇有一个萝卜对着一个坑的意味。

还以前面的函数为例，用下面的方式赋值就不用担心顺序问题了。

```
>>> foo(p3=3, p1=10, p2=222)
p1==> 10
p2==> 222
p3==> 3
```

还可以用类似下面的方式，部分参数给予默认的值。

```
>>> def foo(p1, p2=22, p3=33):     #设置了两个参数p2, p3的默认值
...     print "p1==>", p1
...     print "p2==>", p2
...     print "p3==>", p3
...
>>> foo(11)                        #p1=11，其他的参数为默认赋值
p1==> 11
p2==> 22
p3==> 33

>>> foo(11, 222)                   #按照顺序，p2=222，p3依旧维持原默认值
p1==> 11
p2==> 222
p3==> 33
>>> foo(11, 222, 333)
p1==> 11
p2==> 222
p3==> 333
```

```
>>> foo(11, p2=122)
p1==> 11
p2==> 122
p3==> 33

>>> foo(p2=122)                    #p1没有默认值，必须要赋值的，否则报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes at least 1 argument (1 given)
```

def foo(*args)

这种方式适合于不确定参数个数的时候，在参数args前面加一个*，注意，仅加一个。

```
>>> def foo(*args):
...     print args
...
>>> foo("qiwsir.github.io")
('qiwsir.github.io',)
>>> foo("qiwsir.github.io","python")
('qiwsir.github.io', 'python')
```

def foo(args)**

这种方式跟上面的区别在于，必须接收类似arg=val的形式。

```
>>> def foo(**args):
...     print args
...

>>> foo(1, 2, 3)                    #这样就报错了
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 0 arguments (3 given)

>>> foo(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
```

下面写一个综合的，看看以上四种参数传递方法的执行顺序。

```
>>> def foo(x,y=2, *targs, **dargs):
...     print "x==>", x
...     print "y==>", y
...     print "targs_tuple==>", targs
...     print "dargs_dict==>", dargs
```

```
...

>>> foo("1x")
x==> 1x
y==> 2
targs_tuple==> ()
dargs_dict==> {}

>>> foo("1x", "2y")
x==> 1x
y==> 2y
targs_tuple==> ()
dargs_dict==> {}

>>> foo("1x", "2y", "3t1", "3t2")
x==> 1x
y==> 2y
targs_tuple==> ('3t1', '3t2')
dargs_dict==> {}

>>> foo("1x", "2y", "3t1", "3t2", d1="4d1", d2="4d2")
x==> 1x
y==> 2y
targs_tuple==> ('3t1', '3t2')
dargs_dict==> {'d2': '4d2', 'd1': '4d1'}
```

3.4 特殊函数

到目前为止，你已经知道怎么写一个函数了。但是，从“知道”到“做到”还有很长的距离，要自己能够熟练写出函数，还需要读者找机会多练习。

在Python中，除了自己写函数之外，它还提供了一些特有的函数，比如前面已经多次出现的内建函数，还有某些模块中的函数等，但是，看了这里要说的特殊函数之后，你的确会感受到它们的不一样，它们常常被认为是“函数式编程”的代表。

3.4.1 递归

递归不是函数，而是一种思想。之所以放到这里是因为在函数中要用到，所以先行说明此思想，然后讲述几个特殊函数。

什么是递归？

递归，见递归。

这是对“递归”最精简的定义。还可以用另外一种形式定义：

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事。故事是什么呢？从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事。故事是什么呢？从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事。故事是什么呢？……

如果用上面的内容做递归的定义，总感觉有点调侃，来个严肃的（选自维基百科）：

递归（英语：Recursion），又译为递回，在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。

最典型的递归例子之一是斐波那契数列。

根据斐波那契数列的定义，可以直接写成这样的斐波那契数列递

归函数。

```
#!/usr/bin/env python
# coding=utf-8

def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    f = fib(10)
    print f
```

把上述代码保存。这个代码的意图是要得到n＝10的值。运行之：

```
$ python 20401.py
55
```

fib（n-1）＋fib（n-2）就是又调用了这个函数自己，实现递归。为了明确递归的过程，下面走一个计算过程（考虑到次数不能太多，就让n＝3）。

1.n＝3，fib（3），自然要走return fib（3-1）＋fib（3-2）分支。

2.先看fib（3-1），即fib（2）也要走else分支，于是计算fib（2-1）＋fib（2-2）。

3.fib（2-1）即fib（1），在函数中就要走elif分支，返回1，即fib（2-1）＝1。同理，容易得到fib（2-2）＝0。将这两个值返回到上面一步。得到fib（3-1）＝1＋0＝1。

4.再计算fib（3-2），就简单了一些，返回的值是1，即fib（3-2）＝1。

5.最后计算第一步中的结果：fib（3-1）＋fib（3-2）＝1＋1＝2，将计算结果2作为返回值。

从而得到fib（3）的结果是2。

从上面的过程中可以看出，每个递归的过程都是向着最初的已知条件a0＝0和a1＝1方向挺进一步，直到通过这个最底层的条件得到结果，然后再一层一层向上回馈计算结果。

其实，上面的代码有一个问题。因为a0＝0、a1＝1是已知的了，不需要每次都判断一边。所以，还可以优化一下，优化的基本方案就是初始化最初的两个值。

```
#!/usr/bin/env python
# coding=utf-8

meno = {0:0, 1:1}                    #初始化

def fib(n):
    if not n in meno:
        meno[n] = fib(n-1) + fib(n-2)
    return meno[n]

if __name__ == "__main__":
    f = fib(10)
    print f
#运行结果
$ python 20402.py
55
```

以上实现了递归，但是，至少在Python中，递归要慎重使用。因为在一般情况下，递归是能够被迭代或者循环替代的，而且后者的效率常常比递归要高。所以，我的个人建议是，对使用递归要考虑周密，不小心就会永远运行下去。

3.4.2　几个特殊函数

特殊函数的特殊在于跟“函数式编程”扯上了关系。

如果以前没有听过，等你开始进入编程界，也会经常听人说“函数式编程”、“面向对象编程”、“指令式编程”等术语。它们是什么呢？这个话题要从“编程范式”讲起。（以下内容源自维基百科）

编程范型或编程范式（英语：Programming paradigm范即模范之意，范式即模式、方法），是一类典型的编程风格，是指从事软件

工程的一类典型的风格（可以对照方法学）。如：函数式编程、程序编程、面向对象编程、指令式编程等为不同的编程范型。

编程范型提供了（同时决定了）程序员对程序执行的看法。例如，在面向对象编程中，程序员认为程序是一系列相互作用的对象，而在函数式编程中一个程序会被看作是一个无状态的函数计算的串行。

正如软件工程中不同的群体会提倡不同的“方法学”一样，不同的编程语言也会提倡不同的“编程范型”。一些语言是专门为某个特定的范型设计的（如Smalltalk和Java支持面向对象编程，而Haskell和Scheme则支持函数式编程），同时还有另外一些语言支持多种范型（如Ruby、Common Lisp、Python和Oz）。

编程范型和编程语言之间的关系十分复杂，由于一个编程语言可以支持多种范型。例如，C＋＋设计时，支持过程化编程、面向对象编程以及泛型编程。然而，设计师和程序员们要考虑如何使用这些范型元素来构建一个程序。一个人可以用C＋＋写出一个完全过程化的程序，另一个人也可以用C＋＋写出一个纯粹的面向对象程序，甚至还有人可以写出杂糅了两种范型的程序。

建议读者将上面这段话认真读完，不管是理解还是不理解，总能有点感觉的。

正如前面引文中所说的，Python是支持多种范型的语言，可以进行所谓的函数式编程，其突出体现在有这么几个函数：

filter、map、reduce、lambda、yield

有了它们，最大的好处是程序更简洁；没有它们，程序也可以用别的方式实现，只不过可能要多写几行罢了。所以，还是能用则用之吧。更何况，恰当地使用这几个函数，能让别人感觉你更牛。（注：本节不对yield进行介绍，后面介绍。）

1.lambda

lambda函数是一个只用一行就能解决问题的函数，听着是多么诱人呀。看下面的例子：

```
>>> def add(x):
...     x += 3
```

```
...     return x
...
>>> numbers = range(10)
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(add(i))   #调用add()函数，并append到list中
...
>>> new_numbers
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

在这个例子中，add()只是一个中间操作。当然，上面的例子完全可以用别的方式实现。比如：

```
>>> new_numbers = [ i+3 for i in numbers ]
>>> new_numbers
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

首先说明，这种列表解析的方式是非常好的。

但是，我们偏偏要用lambda这个函数替代add（x），如果你和我一样这么偏执，就可以：

```
>>> lam = lambda x: x + 3
>>> n2 = []
>>> for i in numbers:
...     n2.append(lam(i))
...
>>> n2
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

这里的lam就相当于add（x），请对应一下，这一行lambda x:x＋3就完成add（x）的三行，特别是最后返回值。还可以写这样的例子：

```
>>> g = lambda x,y:x+y
>>> g(3,4)
7
>>> (lambda x:x**2)(4)
16
```

通过上面的例子，总结一下lambda函数的使用方法：

- 在lambda后面直接跟变量。
- 变量后面是冒号。
- 冒号后面是表达式，表达式计算结果就是本函数的返回值。

为了简明扼要，用一个式子表示是必要的：

```
lambda arg1, arg2, ...argN : expression using arguments
```

要特别提醒读者：虽然lambda函数可以接收任意多个参数（包括可选参数）并且返回单个表达式的值，但是lambda函数不能包含命令，包含的表达式不能超过一个。不要试图向lambda函数中塞入太多的东西；如果你需要更复杂的东西，应该定义一个普通函数，想让它多长就多长。

就lambda而言，它并没有给程序带来性能上的提升，但带来的是代码的简洁。比如，要打印一个list，里面依次是某个数字的1次方、二次方、三次方、四次方。用lambda可以这样做：

```
>>> lamb = [ lambda x:x, lambda x:x**2, lambda x:x**3, lambda x:x*
>>> for i in lamb:
...     print i(3),
...
3 9 27 81
```

lambda作为一个单行的函数，在编程实践中可以选择使用。

2.map

先看一个例子，还是上面讲述lambda时的第一个例子，用map也能够实现：

```
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> map(add, numbers)                 #只引用函数名称add即可
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

>>> map(lambda x: x+3, numbers)       #用lambda当然可以啦
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

map()是Python的一个内置函数，它的基本样式是：

```
```

```
map(func, seq)
```

func是一个函数，seq是一个序列对象。在执行的时候，序列对象中的每个元素，按照从左到右的顺序依次被取出来，塞入到func函数里面，并将func的返回值依次存到一个列表中。

在应用中，map所能实现的也可以用别的方式实现。比如：

```
>>> items = [1, 2, 3, 4, 5]
>>> squared = []
>>> for i in items:
...     squared.append(i**2)
...
>>> squared
[1, 4, 9, 16, 25]

>>> def sqr(x): return x**2
...
>>> map(sqr, items)
[1, 4, 9, 16, 25]

>>> map(lambda x: x**2, items)
[1, 4, 9, 16, 25]

>>> [ x**2 for x in items ]          #这个我最喜欢了，速度快，可读性
[1, 4, 9, 16, 25]
```

条条大路通罗马，在编程中，以上方法自己根据需要来选用。

在以上感性认识的基础上，再来阅读map()的官方说明，能够更明白一些。

map（function，iterable，...）

Apply function to every item of iterable and return a list of the results.If additional iterable arguments are passed，function must take that many arguments and is applied to the items from all iterables in parallel.If one iterable is shorter than another it is assumed to be extended with None items.If function is None，the identity function is assumed;if there are multiple arguments，map()returns a list consisting of tuples containing the corresponding items from all iterables（a kind of transpose operation）.The iterable arguments may be a sequence or any iterable object;the result is always a list.

理解要点：

- 对iterable中的每个元素，依次应用function的方法（函数）（这本质上就是一个for循环）。
- 将所有结果返回一个列表。
- 如果参数很多，则对那些参数并行执行function。

例如：

```
>>> lst1 = [1,2,3,4,5]
>>> lst2 = [6,7,8,9,0]
>>> map(lambda x, y: x+y, lst1, lst2)
[7, 9, 11, 13, 5]
```

请读者注意了，上面这个例子如果用for循环来写，还不是很难，如果扩展一下，下面的例子用for来改写，就要小心了：

```
>>> lst1 = [1,2,3,4,5]
>>> lst2 = [6,7,8,9,0]
>>> lst3 = [7,8,9,2,1]
>>> map(lambda x, y, z: x+y+z, lst1, lst2, lst3)
[14, 17, 20, 15, 6]
```

这才显示出map的简洁优雅。

```
reduce
```

直接看这个：

```
>>> reduce(lambda x, y : x+y, [1, 2, 3, 4, 5])
15
```

请仔细观察，是否能够看出如何运算的呢？如图3-2所示。

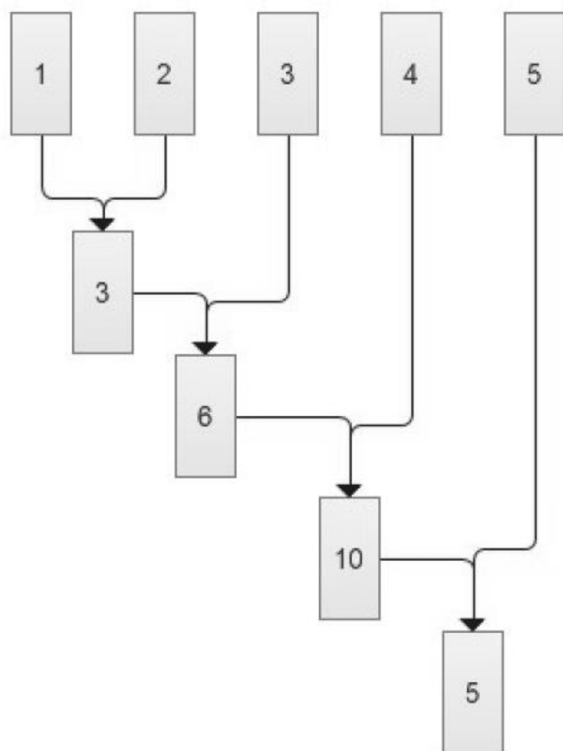


图3-2　运算图

对比一下map()的运算，能更理解reduce()。Map()是上下运算，reduce()是横着逐个元素进行运算。

权威的解释来自官网：

reduce（function，iterable[，initializer]）

Apply function of two arguments cumulatively to the items of iterable，from left to right，so as to reduce the iterable to a single value.For example，reduce（lambda x，y:x+y，[1，2，3，4，5]）calculates（（（（1+2）+3）+4）+5）.The left argument，x，is the accumulated value and the right argument，y，is the update value from the iterable.If the optional initializer is present，it is placed before the items of the iterable in the calculation，and serves as a default when the iterable is empty.If initializer is not given and iterable contains only one item，the first

item is returned.Roughly equivalent to:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no in
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

如果用我们熟悉的for循环来做上面reduce的事情，可以这样来做：

```
>>> lst = range(1, 6)
>>> lst
[1, 2, 3, 4, 5]
>>> r = 0
>>> for i in range(len(lst)):
...     r += lst[i]
...
>>> r
15
```

for是普适的，reduce是简洁的。

为了锻炼思维，看这么一个问题，有两个list，a=[3，9，8，5，2]，b=[1，4，9，2，6]，计算：a[0]*b[0]+a[1]*b[1]+...的结果。

```
>>> a
[3, 9, 8, 5, 2]
>>> b
[1, 4, 9, 2, 6]

>>> zip(a ,b)
[(3, 1), (9, 4), (8, 9), (5, 2), (2, 6)]

>>> sum(x*y for x, y in zip(a, b))          #解析后直接求和
133

>>> new_list = [x*y for x, y in zip(a, b)]    #可以看作是上面方法的分步
```

```
>>> #这样解析也可以：new_tuple = (x*y for x, y in zip(a, b))
>>> new_list
[3, 36, 72, 10, 12]
>>> sum(new_list)                              #或者:sum(new_tuple)
133

>>> reduce(lambda sum, (x, y): sum + x*y, zip(a,b), 0)    #这个方法
133

>>> from operator import add, mul              #耍酷的方法也不止一个
>>> reduce(add,map(mul,a,b))
133

>>> reduce(lambda x,y: x+y, map(lambda x,y: x*y, a,b))  #map,reduc
133
```

如果读者使用的是Python 3，会跟上面有点儿不一样，因为在Python 3中，reduce()已经从全局命名空间中移除，放到了functools模块中，如果要是用，需要用from functools import reduce引入之。

3.filter

filter的中文含义是“过滤器”，在Python中，它起到了过滤器的作用。官方文档中这么说：

filter（function，iterable）

Construct a list from those elements of iterable for which function returns true.iterable may be either a sequence，a container which supports iteration，or an iterator.If iterable is a string or a tuple，the result also has that type;otherwise it is always a list.If function is None，the identity function is assumed，that is，all elements of iterable that are false are removed.

Note that filter（function，iterable）is equivalent to[item for item in iterable if function（item）]if function is not None and[item for item in iterable if item]if function is None.

这次真的不翻译了，而且也不解释要点了。请读者务必自己阅读上面的文字，并且理解其含义。

通过下面的程序代码体会：

```
>>> numbers = range(-5, 5)
>>> numbers
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> filter(lambda x: x>0, numbers)
[1, 2, 3, 4]

>>> [x for x in numbers if x>0]          #与上面那句等效
[1, 2, 3, 4]

>>> filter(lambda c: c!='i', 'qiwsir')
'qwsr'
```

至此，介绍了几个函数，这些函数在对程序的性能提高上并没有显著或者稳定预期，但是，在代码的简洁上是有目共睹的。有时候可以用来秀一秀，以彰显Python的优雅，或者自己耍酷。如何用及怎么用，就看你自己的喜好了。

3.5 练习

仅仅知道了函数的知识是远远不够的，必须要勤练习、多敲代码，才能有体会，才能熟练使用。所以，这里专门设立一节，带领读者做几个练习。

首先声明，以下对每个问题的解决方案，不一定是最优的。如果读者有更好的解决方案，可以分享（到我的网站www.itdiffer.com能够找到各种联系我的方法）。

读者完成下面的练习，请遵守如下规则（全看你的自我控制能力了，自控能力强的胜出，不要欺骗哦，因为上帝在看着你呢）：

- 先根据自己的设想写下代码，然后运行调试，检查得到的结果是否正确。
- 也给出了参考代码，但是，参考代码并不是最终结果。
- 可以在上述基础上对代码进行完善。
- 如果读者愿意，可以将代码提交到github上跟大家分享。

3.5.1 解一元二次方程

解一元二次方程是初中数学中的基本知识，一般来讲解法有公式法、因式分解法等。读者可以根据自己的理解，写一段求解一元二次方程的程序。

最简单的思路就是用公式法求解，这是普适法则。

古巴比伦留下的陶片显示，在大约公元前2000年（2000 BC）古巴比伦的数学家就能解一元二次方程了。在大约公元前480年，中国人已经使用配方法求得了二次方程的正根，但是并没有提出通用的求解方法。公元前300年左右，欧几里得提出了一种更抽象的几何方法求解二次方程。

7世纪印度的婆罗摩笈多（Brahmagupta）是第一位懂得使用代数方程的人，同时容许方程有正负数的根。

11世纪阿拉伯的花拉子密独立地发展了一套公式以求方程的正数解。亚伯拉罕·巴希亚（亦以拉丁文名字萨瓦索达著称）在他的著作《Liber embadorum》中，首次将完整的一元二次方程解法传入欧洲。（源自《维基百科》）

参考代码：

```
#!/usr/bin/env python
# coding=utf-8
"""
    solving a quadratic equation
"""

from __future__ import division
import math

def quadratic_equation(a,b,c):
    delta = b * b - 4 *a * c
    if delta < 0:
        return False
    elif delta == 0:
        return -(b / (2 * a))
    else:
        sqrt_delta = math.sqrt(delta)
        x1 = (-b + sqrt_delta) / (2 * a)
        x2 = (-b - sqrt_delta) / (2 * a)
        return x1, x2

if __name__ == "__main__":
    print "a quadratic equation: x^2 + 2x + 1 = 0"
    coefficients = (1, 2, 1)
    roots = quadratic_equation(*coefficients)
    if roots:
        print "the result is:", roots
    else:
        print "this equation has no solution."
```

保存为20501.py，并运行之：

```
$ python 20501.py
a quadratic equation: x^2 + 2x + 1 = 0
the result is: -1.0
```

能够正常运行，求解方程。

但是，如果再认真思考，会发现上述代码是有很大改进空间的：

1.如果不小心将第一个系数（a）的值输入了0，程序肯定会报错。如何避免之？要记住，任何人的输入都是不可靠的。

2.结果貌似只能是小数，这在某些情况下是近似值，能不能得到以分数形式表示的精确结果呢？

3.复数，Python是可以表示复数的，如果delta＜0，是不是写成复数更好。

读者是否还有其他改进呢？你能不能进行改进，然后跟我和其他朋友一起来分享你的成就呢？

至少要完成上述改进，可能需要其他有关Python的知识，甚至于前面没有介绍。这都不要紧，掌握了基本知识之后，在编程的过程中，就要不断发挥google搜索的优势，让它帮助你找寻完成任务的工具。

Python是一个开发的语言，很多大牛人都写了一些工具让别人使用，减轻了后人的劳动负担，这就是所谓的第三方模块。虽然Python中已经有一些“自带电池”，即默认安装，比如上面程序中用到的math，但是我们还嫌不够。于是有很多第三方的模块来专门解决某个问题。比如，这个解方程问题就可以使用SymPy来解决，当然NumPy也是非常强悍的工具。

3.5.2　统计考试成绩

每次考试之后，教师都要统计考试成绩，一般包括：平均分，以及对所有人按成绩从高到低排队，谁成绩最好，谁成绩最差等。下面的任务就是读者转动脑筋，思考如何用程序实现考试成绩统计。为了简化，以字典形式表示考试成绩记录，例如，{"zhangsan":90，"lisi":78，"wangermazi":39}，当然，也许不止这三项，每个老师所处理的内容都稍有不同，因此字典里的键值对也不一样。

怎么做？

有几种可能要考虑到：

- 最高分或者最低分，可能有人并列。
- 要实现不同长度的字典作为输入值。

- 输出结果中，除了平均分，其他的都要有姓名和分数两项，否则都匿名了，怎么刺激学渣、表扬学霸呢？

不管你是学渣还是学霸，都能学好Python。请思考后敲代码调试你的程序，调试之后再阅读下文。

参考代码：

```
#!/usr/bin/env python
# coding=utf-8
"""
  统计考试成绩
"""
from __future__ import division

def average_score(scores):
    """
        统计平均分.
    """
    score_values = scores.values()
    sum_scores = sum(score_values)
    average = sum_scores / len(score_values)
    return average

def sorted_score(scores):
    """
        对成绩从高到低排队.
    """
    score_lst = [(scores[k], k) for k in scores]
    sort_lst = sorted(score_lst, reverse=True)
    return [(i[1], i[0]) for i in sort_lst]

def max_score(scores):
    """
        成绩最高的姓名和分数.
    """
    lst = sorted_score(scores)          #引用分数排序的函数sorted_sco
    max_score = lst[0][1]
    return [(i[0], i[1]) for i in lst if i[1] == max_score]

def min_score(scores):
    """
        成绩最低的姓名和分数.
    """
    lst = sorted_score(scores)
    min_score = lst[len(lst)-1][1]
    return [(i[0],i[1]) for i in lst if i[1]==min_score]
```

```
if __name__ == "__main__":
    examine_scores = {"google":98, "facebook":99, "baidu":52, "ali

    ave = average_score(examine_scores)
    print "the average score is: ", ave            #平均分

    sor = sorted_score(examine_scores)
    print "list of the scores: ",sor               #成绩表

    xueba = max_score(examine_scores)
    print "Xueba is: ",xueba                       #学霸们

    xuezha = min_score(examine_scores)
    print "Xuzha is: ",xuezha                      #学渣们
```

保存为20502.py，然后运行：

```
$ python 20502.py
the average score is:  80.2222222222
list of the scores:  [('facebook', 99), ('apple', 99), ('amazon',
Xueba is:  [('facebook', 99), ('apple', 99), ('amazon', 99)]
Xuzha is:  [('yahoo', 49)]
```

貌似结果还不错。不过，还有改进余地，看看现实就感觉不怎么友好了。能不能优化一下？当然，里面的函数也不一定是最好的方法，你也可以修改优化。

3.5.3 找质数

这是一个比较常见的题目。我们姑且将范围缩小一下，找出100以内的素数吧。

还是按照前面的惯例，读者先做，然后我提供参考代码，最后优化。

质数（Prime number），又称素数，指在大于1的自然数中，除了1和此整数自身外，无法被其他自然数整除的数（也可定义为只有1和本身两个因数的数）。

哥德巴赫猜想是数论中存在最久的未解问题之一。这个猜想最早出现在1742年普鲁士人克里斯蒂安·哥德巴赫与瑞士数学家莱昂哈德·欧拉的通信中。可以用现代的数学语言陈述哥德巴赫猜想为：“任一

大于2的偶数，都可表示成两个质数之和。”。哥德巴赫猜想在提出后的很长一段时间内毫无进展，直到20世纪20年代，数学家从组合数学与解析数论两方面分别提出了解决的思路，并在其后的半个世纪里取得了一系列突破。目前最好的结果是陈景润在1973年发表的陈氏定理（也被称为“1＋2”）。（源自《维基百科》）

对这个练习，我的思路是先做一个函数，用它来判断某个整数是否是质数，然后循环即可。

参考代码：

```
#!/usr/bin/env python
# coding=utf-8
"""
    寻找质数
"""

import math

def is_prime(n):
    """
        判断一个数是否是质数
    """
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n) + 1)):
        if n % i == 0:
            return False
    return True

if __name__ == "__main__":
    primes = [i for i in range(2,100) if is_prime(i)]    #从2开始，
    print primes
```

代码保存后运行：

```
$ python 20503.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
```

打印出了100以内的质数。

你或许也发现了这个程序需要进一步优化的地方，另外，关于判断质数的方法还有好多种，读者可以自己创造或者从网上搜索一些，拓展思路。

3.5.4 编写函数的注意事项

编写函数，在开发实践中是非常必要和常见的，一般情况，你写的函数应该是：

- 尽量不要使用全局变量。
- 如果参数是可变类型数据，则在函数内不要修改它。
- 每个函数的功能和目标要单纯，不要试图一个函数做很多事情。
- 函数的代码行数尽量少。
- 函数的独立性越强越好，不要跟其他的外部东西产生关联。

第2季　进阶

随着学习的深入，你一定体会到了编程的乐趣，同样也越来越多地听到一些感觉有点“莫名其妙”的名词，这些名词常常是对“某个对象的行为或现状”的概括，这种概括需要学习者用“抽象”的思维方法理解。

从本季开始，我们将共同涉足于此类内容。“类”在编程中不断被用到，所以，它成为了本季的主角。本季中其他内容都是对第1季知识的综合应用，从本季开始，你所敲的代码就越来越接近于开发实践了。自古以来，“行百里路者半九十”，若能够坚持读完本季的内容，那么在Python语言学习上至少已经成为另外的10%。

第4章　类

类是OOP中的重要概念，也是学习Python的一个跨越。总有不少初学者，学到这里就感到迷惑。为此，建议学习者从本章开始一边看书一边敲代码，然后反复推敲，不要追求速度，并且要多上网搜索，理解其中含义。最终的胜利是属于你的。

类也是对世界的抽象结果，所以，读者还要注意“抽象思维”方法在编程中的应用。如何才能具有“抽象”能力呢？一两句话还说不好，但是，勤于练习、惯于思考是必须的。

4.1 基本概念

类，如果是你第一次听到这个词，那么把它作为一个单独的名词会感觉怪怪的，因为在汉语体系中，较常见的是“鸟类”、“人类”等词语，而单独说“类”，总感觉前面缺点修饰成分。其实，它对应的是英文单词class，“类”是由class翻译过来的，你就把它作为一个翻译术语吧。

除了“类”这个术语外，还要经常提到OOP，即面向对象编程（或者“面向对象程序设计”）。

为了理解类和OOP，需要对一些枯燥的名词有所了解。

“行百里路者半九十”，如果读者坚持阅读到本书的这个章节，已经对Python有了初步感受，而“类”就是能够让你在Python学习进程中再上台阶的标志。所以，一定要硬着头皮耐心阅读下去。

所谓“术语”，可以粗浅地理解为某个领域的“行话”，比如在物理学里面，有专门定义的“质量”、“位移”、“速度”等，这些术语有的跟日常生活中的俗称名字貌似一样，但是所指有所不同。

“术语”的主要特征是具有一定的稳定性，并且严谨、简明，不是流行语言。在谈到OOP的时候就会遇到一些术语，需要先明确它们的含义。本节没有特别声明的术语定义均来自维基百科。

4.1.1 问题空间

问题空间是问题解决者对一个问题所达到的全部认识状态，它是由问题解决者利用问题所包含的信息和已贮存的信息主动构成的。

一个问题一般从以下三个方面来定义：

- 初始状态——开始时不完全的信息或令人不满意的状况。
- 目标状态——你希望获得的信息或状态。
- 操作——为了从初始状态迈向目标状态，你可能采取的步骤。

这三个部分加在一起定义了问题空间（problem space）。

4.1.2　对象

对象（object），是面向对象（Object Oriented）中的术语，既表示客观世界问题空间（Namespace）中的某个具体的事物，又表示软件系统解空间中的基本元素。

把object翻译为“对象”是比较抽象的，因此，有人认为，不如翻译为“物件”更好，因为“物件”让人感到一种具体的东西。

这种看法在某些语言中是非常适合的，但是在Python中则无所谓，不管怎样，Python中的一切都是对象，不管是字符串、函数、模块还是类都是对象，“万物皆对象”。

都是对象有什么优势吗？这说明Python天生就是OOP的。也说明Python中的所有东西都能够进行拼凑组合应用，因为对象就是可以拼凑组合应用的。

对于对象这个东西，OOP大师Grandy Booch的定义应该是权威的，相关定义的内容如下。

- 对象：一个对象有自己的状态、行为和唯一的标识；所有相同类型的对象所具有的结构和行为在它们共同的类中被定义。
- 状态（state）：包括这个对象已有的属性（通常是类里面已经定义好的）和对象具有的当前属性值（这些属性往往是动态的）。
- 行为（behavior）：是指一个对象如何影响外界及被外界影响，表现为对象自身状态的改变和信息的传递。
- 标识（identity）：是指一个对象所具有的区别于所有其他对象的属性（本质上指内存中所创建的对象的地址）。

大师的话的确有水平，听起来非常高深。不过，初学者可能理解起来有点困难，下面把大师的话化简一下。

简化之，对象应该具有属性（就是上面的状态，因为属性更常用）、方法（就是上面的行为，方法常被使用）和标识。因为标识是内存中自动完成的，所以，平时不用怎么管理它，主要就是属性和方法。

为了体现“深入浅出”的道理，下面讲一个故事。

既然万物都是对象，那么，某个具体的人也是对象。假设以某个“王美女”为对象说明（这个王美女完全是虚构的，请不要对号入座，更不要想入非非，如果雷同，纯属巧合）。

“王美女”这个对象具有某些特征，眼睛，大；腿，长；皮肤，白。当然，既然是美女，肯定还有别的显明特征，读者可以自己去假设。如果用“对象”的术语来说明，就说这些特征都是她的属性，也就是说属性是一个对象所具有的特征，或曰：是什么。

“王美女”除了具有上面的特征之外，还能做一些事情，比如能唱歌、会吹拉弹唱等。这些都是她能够做的事情。用“对象”的术语来说，这就是她的“方法”，即方法就是对象能够做什么。

任何一个对象都要包括这两部分：属性（是什么）和方法（能做什么）。

4.1.3　面向对象

面向对象，不是说你面对这位“王美女”，是指程序员在开发程序的时候要怎么思考问题，怎么构建程序的事情。

面向对象程序设计（英语：Object-oriented programming，缩写：OOP）是一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例，它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

面向对象程序设计可以看作是一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作是一系列函数的集合，或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其他对象，因此它们都可以被看作是一个小型的“机器”，即对象。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。此外，支持者声称面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。反对者在某些领域对此予以否认。

当我们提到面向对象的时候，它不仅指一种程序设计方法，更多意义上是一种程序开发方式。在这一方面，我们必须了解更多关于面向对象系统分析和面向对象设计（Object Oriented Design，简称OOD）方面的知识。

下面再引用一段来自维基百科中关于OOP的历史。为什么要了解历史？因为历史就是过去的今天，反映了人类的思维发展过程。

面向对象程序设计的雏形，早在1960年的Simula语言中即可发现，当时的程序设计领域正面临着一种危机：在软硬件环境逐渐复杂的情况下，软件如何得到良好的维护？面向对象程序设计在某种程度上通过强调可重复性解决了这一问题。20世纪70年代的Smalltalk语言在面向对象方面堪称经典——以至于30年后的今天依然将这一语言视为面向对象语言的基础。

计算机科学中对象和实例概念的最早萌芽可以追溯到麻省理工学院的PDP-1系统。这一系统大概是最早的基于容量架构（capability based architecture）的实际系统。另外1963年Ivan Sutherland的Sketchpad应用中也蕴含了同样的思想。对象作为编程实体最早于20世纪60年代由Simula 67语言引入思维。Simula这一语言是奥利-约翰·达尔和克利斯登·奈加特在挪威奥斯陆计算机中心为模拟环境而设计的（据说，他们是为了模拟船只而设计的这种语言，并且对不同船只间属性的相互影响感兴趣。他们将不同的船只归纳为不同的类，而每一个对象，基于它的类，可以定义它自己的属性和行为）。这种办法是分析式程序的最早概念体现，在分析式程序中，我们将真实世界的对象映射到抽象的对象叫作“模拟”。Simula不仅引入了“类”的概念，还应用了实例这一思想——这可能是这些概念的最早应用。

20世纪70年代施乐PARC研究所发明的Smalltalk语言将面向对象程序设计的概念定义为：在基础运算中，对对象和消息的广泛应用。Smalltalk的创建者深受Simula 67的主要思想影响，但Smalltalk中的对象是完全动态的——它们可以被创建、修改并销毁，这与Simula中的静态对象有所区别。此外，Smalltalk还引入了继承性的思想，因此一举超越了不可创建实例的程序设计模型和不具备继承性的Simula。此外，Simula 67的思想亦被应用在许多不同的语言中，如Lisp、Pascal。

面向对象程序设计在20世纪80年代成为了一种主导思想，这主要应归功于C＋＋。在图形用户界面（GUI）日渐崛起的情况下，面向对象程序设计很好地适应了潮流。GUI和面向对象程序设计的紧密关联在Mac OS X中可见一斑。Mac OS X是由Objective-C语言写成的，

这一语言是一个仿Smalltalk的C语言扩充版。面向对象程序设计的思想也使事件处理式的程序设计应用地更加广泛（虽然这一概念并非仅存在于面向对象程序设计）。一种说法是，GUI的引入极大地推动了面向对象程序设计的发展。

苏黎世联邦理工学院的尼克劳斯·维尔特和他的同事们对抽象数据和模块化程序设计进行了研究。Modula-2将这些都包括了进去，而Oberon则包括了一种特殊的面向对象方法——不同于Smalltalk与C＋＋。

面向对象的特性也被加入了当时较为流行的语言，如Ada、BASIC、Lisp、Fortran、Pascal等。由于这些语言最初并没有面向对象的设计，故而这种糅合常常会导致兼容性和维护性的问题。与之相反的是，“纯正的”面向对象语言却缺乏一些程序员们赖以生存的特性。在这一大环境下，开发新的语言成为了当务之急。作为先行者，Eiffel成功地解决了这些问题，并成为了当时较受欢迎的语言。

在过去的几年中，Java语言成为了广为应用的语言，除了它与C和C＋＋语法上的近似性。Java的可移植性是它的成功中不可磨灭的一步，因为这一特性已吸引了庞大的程序员群的投入。

在最近的计算机语言发展中，一些既支持面向对象程序设计，又支持面向过程程序设计的语言悄然浮出水面。它们中的佼佼者有Python、Ruby等。

正如面向过程程序设计使得结构化程序设计的技术得以提升，现代的面向对象程序设计方法使得对设计模式的用途、契约式设计和建模语言（如UML）技术也得到了一定提升。

阅读到此，如果读者把前面的文字逐句读过，没有跳跃，则姑且认为你已经对“面向对象”有了一个模糊的认识了。那么，类和OOP有什么关系呢？

4.1.4　类

在面向对象程式设计中，类（class）是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的属性和方法。

类的更严格的定义是由某种特定的元数据所组成的内聚的包。它

描述了一些对象的行为规则，而这些对象就被称为该类的实例。类有接口和结构。接口描述了如何通过方法与类及其实例互操作，而结构描述了一个实例中数据如何划分为多个属性。类是与某个层的对象的最具体的类型。类还可以有运行时表示形式（元对象），它为操作与类相关的元数据提供了运行时支持。

支持类的编程语言在支持与类相关的各种特性方面都多多少少有一些微妙的差异。大多数都支持不同形式的类继承。许多语言还支持提供封装性的特性，比如访问修饰符。类的出现，为面向对象编程的三个最重要的特性（封装性、继承性、多态性）提供了实现的手段。

看到这里，读者或许有一个认识，要OOP编程就得用到类。但是，反过来就不能说了，不是用了类就一定是OOP。

4.1.5　编写类

首先要明确，类是对某一群具有同样属性和方法的对象的抽象。比如这个世界上有很多长翅膀并且会飞的生物，于是聪明的人们就将它们统一称为“鸟”——这就是一个类，虽然它也可以称作“鸟类”。

还是以美女为例子，因为这个例子不仅能让读者阅读时不犯困，还能兴趣盎然。

要定义类，就要抽象，找出共同的方面。

```
class 美女:                  #用class来声明，后面定义的是一个类
    pass
```

从这里开始编写一个类，不过这次暂时不用Python，而是用伪代码，当然，这个代码跟Python相去甚远。如下：

```
class 美女:
    胸围 = 90
    腰围 = 58
    臀围 = 83
    皮肤 = white
    唱歌()
    做饭()
```

定义了一个名称为“美女”的类，其中约定，没有括号的是属性，

带有括号的是方法。这个类仅仅是对美女的通常抽象，并不是某个具体美女。

对于一个具体的美女，比如前面提到的王美女，她是上面所定义的“美女”那个类的具体化，这在编程中称为“美女类”的实例。

```
王美女 = 美女()
```

用这样一种表达方式就是将“美女类”实例化了，对“王美女”这个实例，就可以具体化一些属性，比如胸围；还可以具体实施一些方法，比如做饭。通常可以用这样一种方式表示：

```
a = 王美女.胸围
```

用点号“.”的方式，表示王美女胸围的属性，得到的变量a就是90。另外，还可以通过这种方式给属性赋值，比如

```
王美女.皮肤 = black
```

这样，这个实例（王美女）的皮肤就是黑色的了。

通过实例，也可以访问某个方法，比如：

```
王美女.做饭()
```

这就是在执行一个方法，让王美女这个实例做饭。现在也比较好理解，只有一个具体的实例才能做饭。

4.2　详解类

现在开始不用伪代码了，用真正的Python代码来理解类。当然，例子还是要用读者感兴趣的例子。

4.2.1　新式类和旧式类

因为Python是一个不断发展的高级语言，导致了在Python 2.x的版本中，有“新式类”和“旧式类（也叫做经典类）”之分。新式类是Python 2.2引进的，在此后的版本中，我们一般用的都是新式类。本着知其然还要知其所以然的目的，简单回顾一下两者的差别。

```
>>> class AA:
...     pass
```

这定义了一个非常简单的类，而且是旧式类。至于如何定义类，下面会详细说明。读者姑且认同我刚才建立的名为AA的类，为了简单，这个类内部什么也不做，就是用pass一带而过。但不管怎样它是一个类，而且是一个旧式类。

然后，将这个类实例化：

```
>>> aa = AA()
```

不要忘记实例化的时候类的名称后面有一对括号。接下来做如下操作：

```
>>> type(AA)
<type 'classobj'>
>>> aa.__class__
<class __main__.AA at 0xb71f017c>
>>> type(aa)
<type 'instance'>
```

解读一下上面的含义。

- type（AA）：查看类AA的类型，返回的是'classobj'。
- aa.__class__：aa是一个实例，也是一个对象，每个对象都有__class__属性，用于显示它的类型。这里返回的结果是，从这个结果中可以读出的信息是，aa是类AA的实例，并且类AA在内存中的地址是0xb71f017c。
- type（aa）：是要看实例aa的类型，它显示的结果是instance，意思是告诉我们它的类型是一个实例。

在这里是不是感觉有点不和谐呢？aa.__class__和type（aa）都可以查看对象类型，但是它们显示不一样的结果。比如，查看这个对象：

```
>>> a = 7
>>> a.__class__
<type 'int'>
>>> type(a)
<type 'int'>
```

别忘记了，前面提到过的“万物皆对象”，那么一个整数7也是对象，用两种方式查看，返回的结果一样。为什么到类（严格讲是旧式类）这里，居然返回的结果不一样呢？太不和谐了。

于是乎，就有了新式类，从Python 2.2开始，变成这样了：

```
>>> class BB(object):
...     pass
...
>>> bb = BB()
>>> bb.__class__
<class '__main__.BB'>
>>> type(bb)
<class '__main__.BB'>
```

终于把两者统一起来了，世界和谐了。

这就是新式类和旧式类的不同。

当然，不同点绝非仅仅于此，这里只不过提到一个现在能够理解的不同罢了。另外的不同还在于两者对于多重继承的查找和调用方法不同，旧式类是深度优先，新式类是广度优先。

不管是新式类还是旧式类，都可以通过这样的方法查看它们在内存中的存储空间信息。

```
>>> print aa
<__main__.AA instance at 0xb71efd4c>

>>> print bb
<__main__.BB object at 0xb71efe6c>
```

两个实例分别告诉了我们其是基于谁生成的，不过还是稍有区别。

知道了旧式类和新式类，那么下面的所有内容就都是对新式类而言。“喜新厌旧”不是编程经常干的事情吗？所以，旧式类就不是我们讨论的内容了。

还要注意，如果你用的是Python 3，就不用为新式类和旧式类而担心了，因为在Python 3中压根儿就不存在这个问题。

如何定义新式类呢？

第一种定义方法如同前面那样：

```
>>> class BB(object):
...     pass
...
```

跟旧式类的区别就在于类的名字后面跟上（object），这其实是一种名为“继承”的类的操作，当前的类BB是以类object为上级的（object被称为父类），即BB是继承自类object的新类。在Python 3中，所有的类自然地都是类object的子类，就不用彰显出继承关系了。

第二种定义方法，在类的前面写上：_metaclass_＝＝type，然后定义类的时候，就不需要在名字后面写（object）了。

```
>>> __metaclass__ = type
>>> class CC:
...     pass
...
>>> cc = CC()
>>> cc.__class__
<class '__main__.CC'>
>>> type(cc)
<class '__main__.CC'>
```

两种方法，任你选用，没有优劣之分。

4.2.2 创建类

在一般情况下，一个类不是两三行就能搞定的。所以，下面可能很少使用交互模式了，因为那样一旦有一点错误就前功尽弃。下面改用编辑界面。

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def color(self, color):
        print "%s is %s" % (self.name, color)
```

上面定义的是一个比较常见的类，下面对这个“大众脸”的类进行一一解释。

（1）新式类

metaclass = type，意味着下面的类是新式类。

（2）定义类

class Person，这是在声明创建一个名为“Person”的类。类的名称一般用大写字母开头，这是惯例。如果名称是两个单词，那么两个单词的首字母都要大写，例如class HotPerson，这种命名方法有一个形象的名字，叫作“驼峰式命名”。当然，如果故意不遵循此惯例，也未尝不可，但是，会给别人阅读乃至于自己以后阅读带来麻烦，不要忘记“代码通常是给人看的，只是偶尔让机器执行”。既然大家都是靠右走的，你就别非要在路中间睡觉了。

分别以缩进表示的就是这个类的内容了。其实那些东西看起来并不陌生——就是已经学习过的函数。不过，很多程序员喜欢把类里面

的函数叫作“方法”，就是上节中说到的对象的“方法”。曾看到有人撰文专门分析了“方法”和“函数”的区别。但是，我倒是认为这不重要，重要的是类中所谓“方法”和前面的函数，从数学角度看，丝毫没有区别。所以，你尽可以称之为函数。当然，若听到有人说方法，也不要诧异和糊涂，它们本质是一样的。

需要再次提醒，函数的命名方法是以def发起，并且函数名称首字母不要用大写，可以使用aa_bb的样式，也可以使用aaBb的样式，一切看你的习惯了。

要注意的是，类中的函数（方法）的参数跟以往的参数样式有区别，那就是每个函数必须包括self参数，并且作为默认的第一个参数，这是需要注意的地方。至于它的用途，继续学习即可知道。

（3）初始化

def_init_这个函数一个比较特殊，并且有一个名字，叫作初始化函数（注意，很多教材和资料中把它叫作构造函数，这种说法貌似没有错误，但是从字面意义上看，它对应的含义是初始化，而且在Python中它的作用和其他语言比还不完全一样，因为还有一个_new_的函数是真正的构造。所以，我称之为初始化函数）。它是以两个下画线开始，然后是init，最后以两个下画线结束。

所谓初始化，就是让类有一个基本的面貌。做很多事情都要初始化，让事情有一个具体的起点状态。比如你要喝水，必须先初始化杯子里面有水。在Python的类中，初始化就担负着类似的工作。这个工作是在类被实例化的时候就执行这个函数，从而将初始化的一些属性可以放到这个函数里面。

此例子中的初始化函数就意味着实例化的时候，要给参数name提供一个值，作为类初始化的内容。就是在这个类被实例化的同时，要通过name参数传一个值，这个值被一开始就写入了类和实例中，成为了类和实例的一个属性。比如：

```
girl = Person('canglaoshi')
```

girl是一个实例对象，它有属性和方法，这里仅说属性吧。当通过上面的方式实例化后，就自动执行了初始化函数，让实例girl就具有了name属性。

```
print girl.name
```

执行这句话的结果是打印出canglaoshi。

这就是初始化的功能。简而言之，通过初始化函数，确定了这个实例（类）的“基本属性”。

初始化函数就是一个函数，所以，它的参数设置也符合前面学过的函数参数设置规范。比如：

```
def __init__(self, *args):
    pass
```

这种类型的参数*args和前面讲述的函数参数一样，self这个参数是必须要有的。

很多时候，并不是每次都要从外面传入数据，有时候会把初始化函数的某些参数设置默认值，如果没有新的数据传入，就应用这些默认值。比如：

```
class Person:
    def __init__(self, name, lang="golang", website="www.google.co
        self.name = name
        self.lang = lang
        self.website = website
        self.email = "qiwsir@gmail.com"

laoqi = Person("LaoQi")
info = Person("qiwsir", lang="python", website="qiwsir.github.io")

print "laoqi.name=", laoqi.name
print "info.name=", info.name
print "-------"
print "laoqi.lang=", laoqi.lang
print "info.lang=", info.lang
print "-------"
print "laoqi.website=", laoqi.website
print "info.website=", info.website
```

运行结果：

```
laoqi.name= LaoQi
info.name= qiwsir
-------
laoqi.lang= golang
info.lang= python
-------
```

```
laoqi.website= www.google.com
info.website= qiwsir.github.io
```

在编程界有这样一句话："类是实例工厂"，什么意思呢？生产物品，比如生产电脑，一个工厂可以生产好多电脑，那么，类就能"生产"好多实例，所以，它是"工厂"。比如上面例子中，就有两个实例。

4.2.3 类中的函数（方法）

这里我用"函数"这个术语，指的是类中的那些函数，也把它叫作"方法"。但是这里不打算太深入区分"方法"和"函数"，因为随着读者对编程的熟悉，自然能领悟到。所以，我可能在有的地方承接原来的习惯用"函数"，也可能用"方法"。

再把已经用过的那个类拿出来研究一番。

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def color(self, color):
        print "%s is %s" % (self.name, color)
```

构造函数的后面有两个函数：def getName（self）和def color（self，color），这两个函数和前面的初始化函数有共同的地方，都是以self作为第一个参数。

```
def getName(self):
    return self.name
```

这个函数的作用就是返回在初始化时得到的值，初始化函数中self.name的值能够在这个函数中被使用，其原因就在于此函数中不可缺少的参数self。

```
girl = Person('canglaoshi')
    name = girl.getName()
```

girl.getName()就是调用实例girl的getName()方法（函数）。调用该方法的时候要特别注意，方法名后面的括号不可少，并且括号中不要写参数，在类中的getName（self）函数第一个参数self是默认的，当类实例化之后，调用此函数的时候，第一个参数不需要赋值。那么，变量name的最终结果就是name＝"canglaoshi"。

同样道理，对于方法：

```
def color(self, color):
    print "%s is %s" % (self.name, color)
```

也是在实例化之后调用：

```
girl.color("white")
```

这也是在执行实例化方法，只是由于类中的该方法有两个参数，除了默认的self之外，还有一个color，所以，在调用这个方法的时候，要为后面那个参数传值。

至此，已经将这个典型的类和调用方法分解完毕，把全部代码完整贴出，请读者从头到尾看看，是否理解了每个部分的含义：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def color(self, color):
        print "%s is %s" % (self.name, color)

girl = Person('canglaoshi')                 #实例化
name = girl.getName()                       #调用方法（函数）
print "the person's name is: ", name
girl.color("white")                         #调用方法（函数）
```

```
print "------"
print girl.name                                    #实例的属性
```

保存后，运行得到如下结果：

```
$ python 20701.py
the person's name is:  canglaoshi
canglaoshi is white
------
canglaoshi
```

4.2.4　类和实例

有必要总结一下类和实例的关系。

（1）“类提供默认行为，是实例的工厂”（源自Learning Python），这句话非常经典，一下道破了类和实例的关系。所谓工厂，就是可以用同一个模子做出很多具体的产品。类就是那个模子，实例就是具体的产品。所以，实例是程序处理的实际对象。

（2）类由一些语句组成，但是实例通过调用类生成，每次调用一个类，就得到这个类的新的实例。

（3）命名类必须用class，例如class Person。Class发起了一个可执行的语句，如果执行，就得到一个类对象，并且将这个类对象赋值给对象名（比如Person）。

也许上述比较还不足以让你把类和实例理解透彻，没关系，继续学习，在前进中排除疑惑。

4.2.5　self的作用

类里面的函数，第一个参数是self，而且不能省略。但是在实例化的时候，这个参数不需要写，也不需要为这个参数传值，似乎没有这个参数什么事儿了，真的是这样吗？self是干什么的呢？

self是一个很神奇的参数。

还是以前面的类“Person”为例，在Person实例化的过程中girl＝Person（"canglaoshi"），字符串"canglaoshi"通过初始化函数

（_init_()）的参数已经存入到内存中，并且以Person类型的面貌存在，组成了一个对象，这个对象和变量girl建立引用关系。这个过程也可说成这些数据附加到一个实例上。这样就能够以object.attribute的形式，在程序中任何地方调用某个对象（数据）。例如上面的程序中以girl.name的方式得到"canglaoshi"。这种调用方式，在类和实例中经常使用，点号“.”后面的称之为类或者实例的属性。

这是在程序中，并且是在类的外面。如果在类的里面，想在某个地方使用实例化所传入的数据（"canglaoshi"），怎么办？

在类内部，就是将所有传入的数据都赋给一个变量，通常这个变量的名字是self。注意，这是习惯，而且是共识，所以，你就不要费尽心机另外取别的名字了。

在初始化函数中的第一个参数self就是起到了这个作用——接收实例化过程中传入的所有数据，这些数据是初始化函数的参数导入的。显然，self应该就是一个实例（准确说法是应用实例），因为它所对应的就是具体数据。

如果将上面的类稍加修改，看看效果：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name
        print self
        print type(self)
```

其他部分省略。当初始化的时候，首先要运行构造函数，同时打印新增的两条。结果是：

```
<__main__.Person object at 0xb7282cec>
<class '__main__.Person'>
```

证实了推理，self就是一个实例（准确说是实例的引用变量）。

self这个实例跟前面说的那个girl所引用的实例对象一样，也有属性。那么，接下来就规定其属性和属性对应的数据。上面代码中：

self.name＝name，就是规定了self实例的一个属性，这个属性的名字也叫作name，且属性的值等于初始化函数的参数name所导入的数据。注意，self.name中的name和初始化函数的参数name没有任何关系，它们两个一样，只不过是一种巧合（经常巧合，其实是为了省事和以后识别方便而故意让它们巧合，或者说是写代码的人懒惰，不想另外取名字而已，无他）。当然，如果写成self.xyz＝name，也是可以的。

其实，从效果的角度来理解更简化：类的实例girl对应着self，girl通过self导入实例属性的所有数据。

当然，self的属性数据，也不一定非得由参数传入，也可以在构造函数中自己设定。比如：

```
#!/usr/bin/env python
#coding:utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name
        self.email = "qiwsir@gmail.com"     #这个属性不是通过参数传入的

info = Person("qiwsir")                     #换个字符串和实例化变量
print "info.name=", info.name
print "info.email=", info.email
```

运行结果：

```
info.name= qiwsir
info.email= qiwsir@gmail.com    #打印结果
```

这个例子让我们拓展了对self的认识，它不仅仅是为了在类内部传递参数导入的数据，还能在初始化函数中，通过self.attribute的方式，规定self实例对象的属性，这个属性也是类实例化对象的属性，即作为类通过初始化函数初始化后所具有的属性。所以在实例info中，通过info.email同样能够得到该属性的数据。在这里，就可以把self形象地理解为“内外兼修”了。或者按照前面所提到的，将info和self对应起来，self主内，info主外。

4.2.6　文档字符串

本书中已经强调过写注释的重要性了，同样，在类里面也要写点东西。

在函数里面，可以用三重引号来写说明，在类中也可以，其实在文件开头的部分也能用三重引号写文档字符串说明。这样写的最大好处是能够用help()函数看。

```
"""This is python lesson"""

def start_func(arg):
    """This is a function."""
    pass

class MyClass:
    """Thi is my class."""
    def my_method(self,arg):
        """This is my method."""
        pass
```

这样的文档是必需的。当然，在编程中，有不少地方要用“#”符号来做注释。

4.3 辨析有关概念

在掌握了类的基本知识的基础上，对与类有关的几个概念进行深入辨析，这种辨析的目的在于让读者对类有更深入的了解，诚然，更需要实践练习，通过实践能够深入感悟到内在的奥秘。

4.3.1 类属性和实例属性

一个类实例化后，实例是一个对象，它有属性。不要忘记，Python中的类也是一个对象，且也有属性。所以就有了“类属性”和“实例属性”两个属性。

```
>>> class A(object):
...     x = 7
...
```

在交互模式下，定义一个很简单的类，类中有一个变量x＝7，当然，如果愿意还可以写得复杂点儿。

```
>>> A.x
7
```

在类A中，变量x所引用的对象，能够直接通过类调用。或者说x是类A的属性，这就是所谓的“类属性”。类属性仅限于此——类中的变量还有另外的称呼，如静态数据。

```
>>> foo = A()
>>> foo.x
7
```

将类A实例化，通过实例foo也可以得到属性（foo.x），这个属性叫作“实例属性”。

对于同一属性，可以用类来访问（类属性），在一般情况下，也可以通过实例来访问同样的属性。

但有时候它们有区别。相同的地方好理解，关键是区别才是编程中要注意之处。

```
>>> foo.x += 1
>>> foo.x
8
>>> A.x
7
```

实例属性（foo.x）更新了，类属性（A.x）没有改变。这至少说明，类属性不会被实例属性左右，也可以进一步说明“类属性与实例属性无关”（这句话不能理解为“类属性”和“实例属性”互不相关，如果要这么理解，按照更严格的逻辑，应该还要看看修改A.x变化是否影响到foo.x）。

那么，foo.x＋＝1的本质是什么呢？

其本质是实例foo又建立了一个新的属性，但是这个属性（新的foo.x）居然与原来的属性（旧的foo.x）重名，所以，原来的foo.x就被“遮盖了”，只能访问到新的foo.x，它的值是8。既然新的foo.x“遮盖”了旧的foo.x，如果删除它，旧的就会显现出来。

```
>>> foo.x
8
>>> del foo.x
>>> foo.x
7
```

的确是这样，删除新的foo.x之后，被它覆盖的foo.x值就显示出来了。

此外，还可以通过建立一个不与旧的实例属性重名的实例属性，理解上述过程。

```
>>> foo.y = foo.x + 1
>>> foo.y
8
>>> foo.x
7
```

foo.y就是新建的一个实例属性，它没有影响原来的实例属性foo.x。

前面看到了实例属性不左右类属性，反过来，类属性能否影响实例属性呢？这点似乎应该好解释，因为实例就是通过实例化类实现的，按照推理，实例属性应该受到类属性的影响。

```
>>> A.x += 1
>>> A.x
8
>>> foo.x
8
```

实例属性的值随着类属性的值变化而改变了。

综上，“类属性不受实例属性影响，但实例属性受到类属性左右”，不过，这个结论是有条件的，前面例子中类内的变量应用的是不可变对象（整数）。根据对可变对象和不可变对象的研究经验（可以参考浅拷贝和深拷贝），按照保守主义的原则（什么是保守主义？保守不等于守旧，其他的含义，读者可以查阅有关书籍资料。），还应该考察对象是可变对象的情形，因为可变数据能够进行原地修改，这可能会导致不一样。

```
>>> class B(object):
...     y = [1, 2, 3]
...
```

这次定义的类中，变量引用的是一个可变对象——列表。

```
>>> B.y
[1, 2, 3]
>>> bar = B()
>>> bar.y
[1, 2, 3]

>>> bar.y.append(4)
>>> bar.y
[1, 2, 3, 4]
>>> B.y
[1, 2, 3, 4]

>>> B.y.append("aa")
>>> B.y
[1, 2, 3, 4, 'aa']
>>> bar.y
[1, 2, 3, 4, 'aa']
```

从上面的比较操作中，你能得出什么结论？当类中变量引用的是可变对象时，类属性和实例属性都能直接修改这个对象，从而影响另一方的值。

以保守主义为原则的思维方法胜利了。

继续看类属性和实例属性的区别。

```
>>> foo = A()
>>> dir(foo)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
```

实例化类A，可以查看其所具有的属性，当然，执行dir（A）也是一样的。

```
>>> A.y = "hello"
>>> foo.y
'hello'
```

增加一个类属性，同时在实例属性中也增加了一样的名称和数据的属性。

反过来，如果增加实例属性，会不会也同时增加一个类属性呢？

```
>>> foo.z = "python"
>>> foo.z
'python'
>>> A.z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'A' has no attribute 'z'
```

类并没有收纳通过实例增加的这个属性，这进一步说明，类属性不受实例属性左右。

不管是通过类，还是通过实例，都可以增加和修改属性，其方法就是通过类或者实例的点号操作来实现，即object.attribute，可以实现对属性的修改和增加。

4.3.2 数据流转

在类的应用中，最广泛的是将类实例化，通过实例来执行各种方法（即类里面的函数）。所以，对此过程中的数据流转一定要弄明白。

还是回到前面让你兴趣盎然的那个实例，只做适当修改，请出"canglaoshi"。这里将注释删除，读者是否能够写上必要的注释呢？如果你能把注释写上，就说明已经理解了类的基本结构。

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def breast(self, n):
        self.breast = n

    def color(self, color):
        print "%s is %s" % (self.name, color)

    def how(self):
        print "%s breast is %s" % (self.name, self.breast)

girl = Person('canglaoshi')
girl.breast(90)

girl.color("white")
girl.how()
```

运行后结果：

```
$ python 20701.py
canglaoshi is white
canglaoshi breast is 90
```

一图胜千言，有图有真相。通过图示，我们看一看数据的流转过程，如图4-1所示。

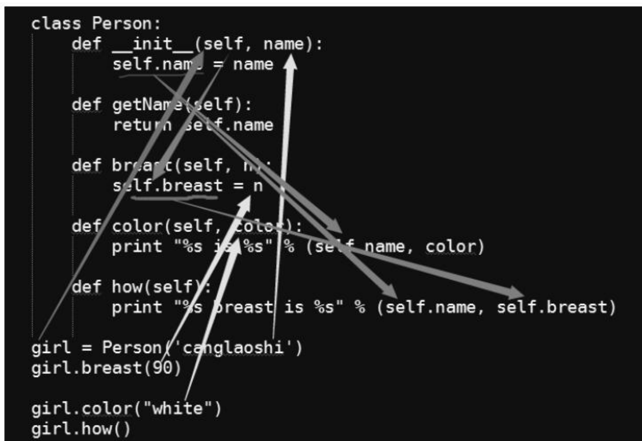


图4-1　数据的流转过程

创建实例girl＝Person（'canglaoshi'），注意观察图上的箭头方向。girl这个实例和Person类中的self对应，这正是应了上节所概括的“实例变量与self对应，实例变量主外，self主内”的结论。"canglaoshi"是一个具体的数据，通过初始化函数中的name参数，传给self.name，你应已知self也是一个实例，可以为它设置属性，self.name就是一个属性，经过初始化函数，这个属性的值由参数name传入，现在就是"canglaoshi"。

在类Person的其他方法中，都是以self为第一个或者唯一一个参数。注意，在Python中，这个参数要显明写上，在类内部的函数的参数中是不能省略的。这就表示所有方法都继承self实例对象，它的属性也被带到每个函数之中。例如在其他函数里面使用self.name即是调用前面已经确定的实例属性数据。当然，在函数中，还可以继续为实例self增加属性，比如self.breast。这样，通过self实例，就实现了数据在类内部的流转。

如果要把数据从类里面传到外面，可以通过return语句实现。如上面例子中所示的getName方法。

实例名称girl和self是对应关系，实际上，在类里面也可以用girl代替self。例如，做如下修改：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type
```

```
class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        #return self.name
        return girl.name          #修改成这样，但是在编程实践中不要这么做。

girl = Person('canglaoshi')
name = girl.getName()
print name
```

运行之后，打印：

```
canglaoshi
```

这个例子说明，在实例化之后，实例变量girl和函数里面的self实例是完全对应的。但是，千万不要用上面修改的方式，因为那样写使类没有独立性，这是大忌。

4.3.3 命名空间

命名空间（namespaces），在研究类或者面向对象编程中，它常常被提到。虽然在函数那部分已经对命名空间有初步解释，但那是在函数的知识范畴中的理解。现在，我们在类的知识范畴中理解“类命名空间”——定义类时，所有位于class语句中的代码都在某个命名空间中执行，即类命名空间。

在研习命名空间以前，请打开Python的交互模式，输入import this，可以看到：

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

这就是所谓的《Python之禅》，请看最后一句“Namespaces are one honking great idea--let's do more of those！”，一般情况下，最后一句都是比较重要的。比如有人做讲座，一大堆PPT一页一页地播放着，口中念念有词，或许对你都没有什么用途，但是当他说出“谢谢聆听”这句话的时候，你一定会从昏昏沉沉中清醒过来，因为他说了最后一句，标志着演说结束。所以，千万要认真听最后一句呀。

《Python之禅》在最后一句中说到了Namespaces，可见命名空间的重要性。

把已经阐述过的命名空间用一句比较学术化的语言概括：命名空间是从所定义的命名到对象的映射集合。

不同的命名空间可以同时存在，彼此相互独立互不干扰。

命名空间因为对象的不同也有所区别，可以分为如下几种。

（1）内置命名空间（Built-in Namespaces）：Python运行起来，它们就存在了。内置函数的命名空间都属于内置命名空间，所以，我们可以在任何程序中直接运行它们，比如前面的id()，不需要做什么操作，拿过来就能直接使用。

（2）全局命名空间（Module:Global Namespaces）：每个模块创建它自己所拥有的全局命名空间，不同模块的全局命名空间彼此独立，不同模块中相同名称的命名空间，也会因为模块的不同而不相互干扰。

（3）本地命名空间（Function&Class:Local Namespaces）：模块中有函数或者类，每个函数或者类所定义的命名空间就是本地命名空间。如果函数返回了结果或者抛出异常，则本地命名空间也结束

了。

如图4-2所示，展示一下上述三种命名空间的关系。

程序在查询上述三种命名空间的时候，按照从里到外的顺序，即：Local Namespaces→Global Namesspaces→Built-in Namesspaces。

```
>>> def foo(num, str):
...     name = "qiwsir"
...     print locals()
...
>>> foo(221, "qiwsir.github.io")
{'num': 221, 'name': 'qiwsir', 'str': 'qiwsir.github.io'}
>>>
```



图4-2　三种命名空间的关系

这是一个访问本地命名空间的方法，用print locals()完成，从这个结果中不难看出，所谓的命名空间中的数据存储结构和dictionary

是一样的。

根据习惯，如果访问全局命名空间，可以使用print globals()。

4.3.4 作用域

作用域是指Python程序可以直接访问到的命名空间。“直接访问”在这里意味着访问命名空间中的命名时无须加入附加的修饰符。

程序也是按照搜索命名空间的顺序，搜索相应空间的能够访问到的作用域。

```
def outer_foo():
    b = 20
    def inner_foo():
        c = 30
a = 10
```

假如我现在位于inner_foo()函数内，那么c对我来讲就在本地作用域，而b和a就不是。如果我在inner_foo()内再做：b＝50，这其实是在本地命名空间内新创建了对象，和上一层中的b＝20毫不相干。可以看下面的例子：

```
#!/usr/bin/env python
#coding:utf-8

def outer_foo():
    a = 10
    def inner_foo():
        a = 20
        print "inner_foo, a=", a        #a=20

    inner_foo()
    print "outer_foo, a=", a            #a=10

a = 30
outer_foo()
print "a=", a                           #a=30
```

运行结果：

```
inner_foo, a= 20
```

```
outer_foo, a= 10
a= 30
```

如果要将某个变量在任何地方都使用，且能够关联，那么在函数内就使用global声明，其实就是曾经讲过的全局变量。

4.4 继承

继承是非常重要的，因为继承让我们能够延续以前的东西，比如“龙生龙、凤生凤、老鼠的儿子会打洞”是基因继承结果，除了生物方面的继承，在现实生活中，“继承”意味着一个人从另外一个人那里得到了一些什么，比如继承革命先烈的光荣传统等。总之，“继承”之后，自己就在所继承的方面省力气，不用劳神费心就能轻松得到。

但是，高级编程语言中的“继承”，跟通常理解的继承会有所不同。“继承”在高级编程语言中是一个非常重要的概念。虽然不用继承一样能够编写程序，但是，当我们追求程序的更高阶层时，继承的作用就显现出来了。

继承（Inheritance）是面向对象软件技术当中的一个概念。如果一个类别A“继承”自另一个类别B，就把这个A称为“B的子类别”，而把B称为“A的父类别”，也可以称“B是A的超类”。

继承可以使得子类别具有父类别的各种属性和方法，而不需要再次编写相同的代码。在令子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能。另外，为子类别追加新的属性和方法也是常见的做法。（源自维基百科）

由上面对继承的表述，简单总结出继承的意图或者好处：

（1）可以实现代码重用，但不是仅仅实现代码重用，有时候根本就没有重用。

（2）实现属性和方法继承。

诚然，以上也不是全部，随着后续学习，对继承的认识会更深刻。好友“令狐虫”曾经这样总结继承：

从技术上说，OOP里继承最主要的用途是实现多态。对于多态而言，重要的是接口继承性，属性和行为是否存在继承性，这是不一定的。事实上，大量工程实践表明，重度的行为继承会导致系统过度复杂和臃肿，反而会降低灵活性。因此现在比较提倡的是基于接口的轻度继承理念。这种模型里因为父类（接口类）完全没有代码，因此根

本谈不上什么代码复用。

在Python里，因为存在Duck Type，接口定义的重要性大大降低，继承的作用也进一步被削弱了。

另外，从逻辑上说，继承的目的也不是为了复用代码，而是为了理顺关系。

或许读者感觉比较高深，没关系，随着你对实践经验的积累，也能对这个问题有自己独到的见解。

或许你也要问我的观点是什么，我的观点就是：走着瞧！怎么理解？继续向下看，只有你先深入这个问题，才能跳到更高层看这个问题。小马过河的故事还记得吧？只有亲自走入河水中，才知道河水的深浅。

对于Python中的继承，前面一直在使用，那就是我们写的类都是新式类，所有新式类都是继承自object类。不要忘记，新式类的一种写法：

```
class NewStyle(object):
pass
```

这就是典型的继承。

4.4.1 基本概念

在编辑器中把这些代码敲出来。

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def speak(self):
        print "I love you."

    def setHeight(self):
        print "The height is: 1.60m."

    def breast(self, n):
```

```
        print "My breast is: ",n

class Girl(Person):
    def setHeight(self):
        print "The height is:1.70m ."

if __name__ == "__main__":
    cang = Girl()
    cang.setHeight()
    cang.speak()
    cang.breast(90)
```

上面这个程序，保存之后运行：

```
$ python 20901.py
The height is:1.70m .
I love you.
My breast is:  90
```

对以上程序进行解释，从中体会继承的概念和方法。

首先定义了一个类Person，在这个类中定义了三个方法。注意，没有定义初始化函数，初始化函数在类中不是必须的。

然后又定义了一个类girl，这个类的名字后面的括号中是上一个类的名字，这就意味着girl继承了Person，girl是Person的子类，Person是girl的父类。

既然是继承了Person，那么girl就拥有了Person中的全部方法和属性（上面的例子没有列出属性）。但是，如果girl里面有一个和Person同样名称的方法，那么就把Person中的同一个方法遮盖住了，显示的是girl中的方法，这叫作方法的重写。

实例化类girl之后，执行实例方法cang.setHeight()，由于在类girl中重写了setHeight方法，那么Person中的那个方法就不显作用了，在这个实例方法中执行的是类girl中的setHeight方法。

虽然在类girl中没有看到speak方法，但是因为它继承了Person，所以cang.speak()就执行类Person中的方法。同理cang.breast（90），它们就好像是在类girl里面已经写了这两个方法一样。

4.4.2 多重继承

所谓多重继承就是指某一个类所继承的父类，不止一个，而是多个。比如：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def eye(self):
        print "two eyes"

    def breast(self, n):
        print "The breast is: ",n

class Girl:
    age = 28
    def color(self):
        print "The girl is white"

class HotGirl(Person, Girl):
    pass

if __name__ == "__main__":
    kong = HotGirl()
    kong.eye()
    kong.breast(90)
    kong.color()
    print kong.age
```

在这个程序中，前面有两个类：Person和girl，然后第三个类HotGirl继承了这两个类，注意观察继承方法，就是在类的名字后面的括号中把所继承的两个类的名字写上。但是第三个类中什么方法也没有。

然后实例化类HotGirl，既然继承了上面的两个类，那么那两个类的方法就都能够拿过来使用。保存程序，运行一下看看：

```
$ python 20902.py
two eyes
The breast is:  90
The girl is white
28
```

值得注意的是，在类girl中，有一个age＝28，在对HotGirl实例化之后，因为继承的原因，这个类属性也被继承到HotGirl中，因此通过实例属性kong.age一样能够得到该数据。

所谓继承，听起来玄乎，实际上没有那么复杂，核心特征是将父类的方法和属性全部承接到子类中；如果子类重写了父类的方法，就使用子类的该方法，父类的方法被遮盖。

4.4.3 多重继承的顺序

学习多重继承的顺序很有必要。比如，如果一个类继承了两个父类，并且两个父类有同样的方法或者属性，那么在实例化子类后，调用哪个父类的方法和属性呢？编造一个没有实际意义，纯粹为了回答这个问题的程序，体会一下多重继承的顺序。

```
#!/usr/bin/env python
# coding=utf-8

class K1(object):
    def foo(self):
        print "K1-foo"

class K2(object):
    def foo(self):
        print "K2-foo"
    def bar(self):
        print "K2-bar"

class J1(K1, K2):
    pass

class J2(K1, K2):
    def bar(self):
        print "J2-bar"

class C(J1, J2):
    pass

if __name__ == "__main__":
    print C.__mro__
    m = C()
    m.foo()
    m.bar()
```

这段代码，保存后运行：

```
$ python 20904.py
    (<class '__main__.C'>, <class '__main__.J1'>, <class '__main__
K1-foo
J2-bar
```

代码中的print C.__mro__是要打印出类的继承顺序。如果要执行foo()方法，首先看J1，没有，看J2，还没有，看J1里面的K1，有了就执行，即C==>J1==>J2==>K1；bar()也是按照这个顺序，在J2中就找到了一个。

这种对继承属性和方法搜索的顺序称之为“广度优先”。

在新式类中，以及python3.x的类中，都是按照“广度优先”原则搜寻属性和方法的。

但是，在旧式类中是按照“深度优先”的顺序的。因为旧式类很少被用到，所以不举例。读者可以自己模仿上面代码，探索旧式类的“深度优先”含义。

4.4.4 super函数

初始化函数的继承跟一般方法的继承还有点不同，可以看下面的例子：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self):
        self.height = 160

    def about(self, name):
        print "{} is about {}".format(name, self.height)

class Girl(Person):
    def __init__(self):
        self.breast = 90

    def about(self, name):
```

```
        print "{} is a hot girl, she is about {}, and her breast i

if __name__ == "__main__":
    cang = Girl()
    cang.about("canglaoshi")
```

在上面这段程序中，类girl继承了类Person。在类girl中，初始化设置了self.breast = 90，由于继承了Person，按照前面的经验，Person的初始化函数中的self.height = 160也应该被Girl所继承过来。然后在重写的about方法中调用self.height。

实例化类girl，并执行cang.about（"canglaoshi"），试图打印出一句话canglaoshi is a hot girl，she is about 160，and her bereast is 90。

保存程序，运行之，看看结果是否如所愿。

```
$ python 20903.py
Traceback (most recent call last):
  File "20903.py", line 22, in <module>
    cang.about("canglaoshi")
  File "20903.py", line 18, in about
    print "{} is a hot girl, she is about {}, and her breast is {}
  AttributeError: 'Girl' object has no attribute 'height'
```

报错！

程序员有一句名言：不求最好，但求报错。

报错不是坏事，而是我们长经验的时候，是在告诉我们，那么做不对。

重要的是看报错信息，就是我们要打印的那句话出问题了，报错信息显示self.height是不存在的，也就是说类girl没有从Person中继承过来这个属性。

原因是什么？仔细观察类girl会发现，除了刚才强调的about方法重写了，_init_方法也被重写了。不要觉得它的名字模样奇怪，就不把它看作类中的方法（函数），它跟类Person中的_init_重名了，同样是重写了那个初始化函数，而girl中的_init_中根本就没有关于self.height的任何信息。

这就提出了一个问题：因为在子类中重写了某个方法之后，父类

中同样的方法被遮盖了，那么如何再把父类的该方法调出来使用呢？纵然被遮盖了，应该还是存在的，不要浪费了呀。

Python中有这样一种被提倡的方法：super函数。

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self):
        self.height = 160

    def about(self, name):
        print "{} is about {}".format(name, self.height)

class Girl(Person):
    def __init__(self):
        super(Girl, self).__init__()
        self.breast = 90

    def about(self, name):
        print "{} is a hot girl, she is about {}, and her breast i
        super(Girl, self).about(name)

if __name__ == "__main__":
    cang = Girl()
    cang.about("canglaoshi")
```

在子类中，_init_方法重写了，为了调用父类同方法，使用super（Girl，self）._init_()的方式。super函数的参数，第一个是当前子类的类名字，第二个是self，然后是点号，点号后面是所要调用的父类的方法。同样在子类重写的about方法中，也可以调用父类的about方法。

执行结果：

```
$ python 20903.py
canglaoshi is a hot girl, she is about 160, and her breast is 90
canglaoshi is about 160
```

最后要注意：super函数仅仅适用于新式类。当然，你使用的一定是新式类，因为“喜新厌旧”是程序员的嗜好。

如果你用的是Python3.x，则使用super函数的形式稍微不同。怎么个不同呢？请你认真阅读下面的“拓展阅读”中的资料（不要责备我不说，我是在告诉你一种非常好的学习方法——看别人提出的问题和解答，因为那些问题也是你的问题）。

4.5 方法

在前面讨论类的时候，“方法”这个词语间或出现，并且还说了不用区分“方法”和“函数”，只要知道所指是什么就可以了。不过，从本节开始，我们要对糊涂的地方稍微澄清一下，并且对“方法”做个深入的探究。

在程序中最常见的是实例化类，通过实例来调用类的方法，对以往的经验稍加概括：

（1）方法是类内部定义函数，只不过这个函数的第一个参数是self（可以认为方法是类属性，但不是实例属性）。

（2）必须将类实例化之后，才能通过实例调用该类的方法。调用的时候在方法后面要有括号（括号中默认有self参数，但是不写出来）。

通过实例调用方法，我们称这个方法绑定在实例上。

4.5.1 绑定方法

调用绑定方法，其实一直在这样做，司空见惯。比如：

```
class Person(object):
    def foo(self):
        pass
```

如果要调用Person.foo()方法，必须：

```
pp = Person()             #实例化
pp.foo()
```

这样就实现了方法和实例的绑定，于是通过pp.foo()即可调用该方法。

4.5.2 非绑定方法

还记得super函数吗？为了描述方便，把代码复制过来：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self):
        self.height = 160

    def about(self, name):
        print "{} is about {}".format(name, self.height)

class Girl(Person):
    def __init__(self):
        super(Girl, self).__init__()
        self.breast = 90

    def about(self, name):
        print "{} is a hot girl, she is about {}, and her breast i
        super(Girl, self).about(name)

if __name__ == "__main__":
    cang = Girl()
    cang.about("canglaoshi")
```

在子类girl中，因为重写了父类的_init_方法，如果要调用父类该方法，不得不使用super（Girl，self）._init_()调用父类中因为子类方法重写而被遮蔽的同名方法。

在子类中，父类的方法就是非绑定方法，因为在子类中，没有建立父类的实例，却要用父类的方法。对于这种非绑定方法的调用，还有一种方式，但现在已经较少使用了，因为有了super函数，为了方便读者看其他有关代码，还是要简要说明一下。

例如在上面的代码中，在类girl中想调用父类Person的初始化函数，则需要在子类中写上这么一行：

```
Person.__init__(self)
```

这不是通过实例调用的，而是通过类Person实现了对

init（self）的调用。这就是调用非绑定方法的用途。但是，这种方法已经被super函数取代，所以，如果读者在编程中遇到类似情况，推荐使用super函数。

4.5.3　静态方法和类方法

类的方法第一个参数必须是self，并且如果要调用类的方法，要通过类的实例，即方法绑定实例后才能由实例调用。如果不绑定，一般在继承关系的类之间，可以用super函数等方法调用。

静态方法和类方法是什么意思呢？跟前面所说的调用类方法有什么关系吗？先看代码：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class StaticMethod:
    @staticmethod
    def foo():
        print "This is static method foo()."

class ClassMethod:
    @classmethod
    def bar(cls):
        print "This is class method bar()."
        print "bar() is part of class:", cls.__name__

if __name__ == "__main__":
    static_foo = StaticMethod()           #实例化
    static_foo.foo()                      #实例调用静态方法
    StaticMethod.foo()                    #通过类来调用静态方法
    print "********"
    class_bar = ClassMethod()
    class_bar.bar()
    ClassMethod.bar()
```

对于这部分代码，有一处非常特别，那就是包含了“@”符号（带@符号的东西是很神奇的）。在Python中：

（1）@staticmethod表示下面的方法是静态方法。

（2）@classmethod表示下面的方法是类方法。

是不是感到有点莫名其妙？少安毋躁，一个一个解释。

先看静态方法，虽然名为静态方法，但也是方法，所以，依然用def语句来定义。需要注意的是文件名后面的括号内没有self，这和前面定义的类中的方法不同，也正是因为这个不同，才给它另外取了一个名字叫作静态方法，否则不就"泯然众人矣"。如果没有self，那么也就无法访问实例变量、类和实例的属性了，因为它们都是借助self来传递数据的。

再看类方法，同样也具有一般方法的特点，区别也在参数上。类方法的参数也没有self，但是必须有cls这个参数。在类方法中，能够访问类属性，但是不能访问实例属性（读者可以自行设计代码检验之）。

明确两种方法之后，继续探究如何调用。两种方法都可以通过实例调用，即绑定实例。也可以通过类来调用，即StaticMethod.foo()这样的形式，这也是区别一般方法的地方，一般方法必须通过绑定实例调用。

上述代码运行结果：

```
$ python 21001.py
This is static method foo().
This is static method foo().
********
This is class method bar().
bar() is part of class: ClassMethod
This is class method bar().
bar() is part of class: ClassMethod
```

这是关于静态方法和类方法的简要介绍。

正当我思考如何讲解地更深入一点的时候，想起了以往看过的一篇文章，觉得人家讲得非常到位。文章标题是：《python中的staticmethod和classmethod的差异》，原载于：www.pythoncentral.io/difference-between-staticmethod-and-classmethod-in-python/。此地址需要你准备梯子才能浏览。后经国人翻译，地址是：http://www.wklken.me/posts/2013/12/22/difference-between-staticmethod-and-classmethod-in-python.html。

4.6 多态和封装

“继承”是类的一个重要特征，在编程中用途很多，虽然在某些具体、细节的层面还有一些不同的看法，但是，这里要说的“多态”和“封装”无论是在理解上还是在实践上都是有争议的话题。所谓争议，多来自于对同一个现象不同角度的理解，特别是有不少经验丰富的程序员，还从其他语言的角度来诠释Python的多态等。不管有多少不同的理解方式，我们都要对这两个东西有所了解，因为它们是你编程水平进阶的必需。

4.6.1 多态

到网上搜索“多态”，仁者见仁智者见智。Python中关于多态的基本体现，可以通过下面的方式来理解。

```
>>> "This is a book".count("s")
2
>>> [1,2,4,3,5,3].count(3)
2
```

count()函数的作用是数一数某个元素在对象中出现的次数。从例子中可以看出，我们并没有限定count的参数所引入的值应该是什么类型的。类似的例子还有：

```
>>> f = lambda x, y: x + y
```

还记得这个lambda函数吗？

```
>>> f(2, 3)
5
>>> f("qiw", "sir")
'qiwsir'
>>> f(["python", "java"], ["c++", "lisp"])
['python', 'java', 'c++', 'lisp']
```

在这个lambda函数中，我们没有限制应该传入什么类型的对象

（或者说数据、值），也一定不能限制，因为如果限制了，就不是pythonic了。也就是说，允许给参数传任意类型的数据，并返回相应的结果，至于是否报错，则取决于“+”的能力范围。这就是“多态”的表现。

“多态”是否能正确表达，不是通过限制传入的对象类型实现，而是这样处理：

```
>>> f("qiw", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
TypeError: cannot concatenate 'str' and 'int' objects
```

这个例子中，把判断两个对象是否能够相加的任务交给了“+”，不是放在入口处判断类型是否为字符串或者数字。

申明，本书由于无意对概念进行讨论，所以不进行这方面的深入探索，仅仅是告诉各位读者相关信息。并且，既然大多数程序员都在讨论多态，那么我们就按照大多数人说的去介绍。

“多态”（Polymorphism），维基百科中对此有详细解释说明。

多型（英语：Polymorphism），是指面向对象程序执行时，相同的信息可能会送给多个不同的类别对象，系统可依据对象所属类别，引发对应类别的方法而有不同的行为。简单来说，所谓多型意指相同的信息给予不同的对象会引发不同的动作。

简化的说法就是“有多种形式”，就算不知道变量（参数）所引用的对象类型，也一样能进行操作，来者不拒，比如上面显示的例子。在Python中，更为pythonic的做法是根本就不进行类型检验。

例如著名的repr()函数，它能够针对输入的任何对象返回一个字符串，这就是多态的代表之一。

```
>>> repr([1, 2, 3])
'[1, 2, 3]'
>>> repr(1)
'1'
>>> repr({"lang": "python"})
"{'lang': 'python'}"
```

使用它写一个小函数，还是作为多态举例。

```
>>> def length(x):
...     print "The length of", repr(x), "is", len(x)
...
>>> length("how are you")
The length of 'how are you' is 11
>>> length([1, 2, 3])
The length of [1, 2, 3] is 3
>>> length({"lang":"python","book":"itdiffer.com"})
The length of {'lang': 'python', 'book': 'itdiffer.com'} is 2
```

不过，多态也不是万能的，如果这样做：

```
>>> length(7)
The length of 7 is
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in length
TypeError: object of type 'int' has no len()
```

报错了。看错误提示，明确告诉了我们“object of type'int'has no len()”，也就是说，函数length()中的len()会对传入的对象进行检验，如果不符合要求，就会报错，使用者可以根据报错信息对传入的对象类型进行调整。

在诸多介绍多态的文章中都会有关于“猫和狗”的例子。这里也将代码贴出来，读者去体会所谓多态体现。其实，如果你进入了Python的语境，有时不经意间就在应用多态特性。

```
#!/usr/bin/env python
# coding=utf-8
'''
    the code is from: http://zetcode.com/lang/python/oop/
'''

__metaclass__ = type

class Animal:
    def __init__(self, name = ""):
        self.name = name
    def talk(self):
        pass

class Cat(Animal):
```

```
    def talk(self):
        print "Meow!"

class Dog(Animal):
    def talk(self):
        print "Woof!"

a = Animal()
a.talk()

c = Cat("Missy")
c.talk()

d = Dog("Rocky")
d.talk()
```

保存后运行之：

```
$ python 21101.py
Meow!
Woof!
```

代码中有Cat和Dog两个类，都继承了类Animal，它们都有talk()方法，输入不同的动物名称，会得出相应的结果。

关于多态，有一个被称作“鸭子类型”（duck typeing）的东西，其含义在维基百科中被表述为：

在程序设计中，鸭子类型（英语：duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于James Whitcomb Riley提出的鸭子测试，“鸭子测试”可以这样表述：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

最后要提示读者，类型检查是毁掉多态的利器，比如type、isinstance以及isubclass函数，所以，一定要慎用这些类型检查函数。

4.6.2 封装和私有化

在正式介绍封装之前，先讲个笑话。

某软件公司老板号称自己懂技术。一次有一个项目要交付给客户，他不想让客户知道实现某些功能的代码，但是交付的时候必须要给人家代码。于是该老板就告诉程序员，“你们把那部分核心代码封装一下”。程序员听完迷茫了。

很多人没有笑，因为不明白说的是什么，不知道你有没有笑。这种幽默唯一的价值在于提到了一个词语“封装”。

“封装”是不是把代码写到某个东西里面，“人”在编辑器中打开也看不到呢？不是，除非你的显示器坏了。

在程序设计中，封装（Encapsulation）是对对象（object）的一种抽象，即将某些部分隐藏起来，在程序外部看不到，无法调用（不是人用眼睛看不到那个代码，除非用某种加密或者混淆方法，造成显示的是一堆混乱的代码，但这不是封装）。

要了解封装离不开“私有化”，就是将类或者函数中的某些属性限制在某个区域之内，外部无法调用，所以先说“私有化”。

“私有化”，顾名思义，就是将某个对象（这个对象可以是你认为的东西）限制在某个自己认定的范围内。比如，某国经济实行私有化，就是将经济体系中组成部分分别纳入到个人权限范畴，而不是放在众多个人无法触及的领域（那是公有），或者说，私有化就是产权清晰。与之对应的是“公有”。

Python中私有化的方法也比较简单，就是在准备私有化的属性（包括方法、数据）名字前面加双下画线。例如：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class ProtectMe:
    def __init__(self):
        self.me = "qiwsir"
        self.__name = "kivi"

    def __python(self):
        print "I love Python."

    def code(self):
        print "Which language do you like?"
        self.__python()
```

```
if __name__ == "__main__":
    p = ProtectMe()
    print p.me
    print p.__name
```

运行一下，看看效果：

```
$ python 21102.py
qiwsir
Traceback (most recent call last):
  File "21102.py", line 21, in <module>
    print p.__name
AttributeError: 'ProtectMe' object has no attribute '__name'
```

查看报错信息，告诉我们没有`_name`那个属性。果然隐藏了，在类的外面无法调用。再试试类里面的那个code()是否可以使用？把该程序做适当修改。

```
if __name__ == "__main__":
        p = ProtectMe()
        p.code()
p.__python()
```

修改好之后保存。其中p.code()的意图是要打印出两句话："Which language do you like?"和"I love Python."，code()方法和私有化的_python()方法在同一个类中，按照私有化的含义，在类里面应该是可以调用的。而p._python()试图通过实例在类的外面调用它。看看效果：

```
$ python 21102.py
Which language do you like?
I love Python.
Traceback (most recent call last):
  File "21102.py", line 23, in <module>
    p.__python()
AttributeError: 'ProtectMe' object has no attribute '__python'
```

如愿以偿，该调用的调用了，该隐藏的隐藏了。

用上面的方法的确做到了封装。但是，如果要调用那些私有属性怎么办？

可以使用property函数。请看下面的例子：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class ProtectMe:
    def __init__(self):
        self.me = "qiwsir"
        self.__name = "kivi"

    @property
    def name(self):
        return self.__name

if __name__ == "__main__":
    p = ProtectMe()
    print p.name
```

运行结果：

```
$ python 21102.py
kivi
```

从上面可以看出，用了@property之后，再调用那个方法的时候，用p.name的形式，就好像在调用以往非私有化属性一样。

看来，封装的确不是“让人看不见”。

4.7 特殊属性和方法

在任何类中，都有一些特殊的属性和方法，它们的特殊性从表观就能看出来，通常是用双画线“_”开头和结尾。本书中把它们归类为特殊的属性和方法，之所以特殊，是因为它们跟你自己写的或者其他不是以“_”开头和结尾的属性、方法有所差异。或许你从事一般开发的项目时，对这些属性和方法使用得不多，但是我认为也是有必要了解的，因为这是你“From Beginner to Master”过程中必须要迈出的一步。知道有这一步，或许会对你的项目有帮助。俗话说“艺不压身”，还是认真了解为好。

4.7.1 _dict_

要访问类或者实例的属性必须通过“object.attribute”的方式，这是我们已经熟知的了。在这个认知的基础上，请思考：类或者实例属性在Python中是怎么存储的？如何修改、增加、删除属性，以及我们能不能控制这些属性？下面就一一道来。

```
>>> class A(object):
...     pass
...

>>> a = A()
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
```

用dir()能够查看类的属性和方法，从上面的结果中可以看出，数量不少，因为我们写的那个类里面只有pass，所以在列出的结果中，都是以“_”开头和结尾的，这些都是所谓的特殊属性和方法。

从众多的内容中寻觅出_dict_，之所以选它，是因为_dict_保存了某些机密。

```
>>> class Spring(object):
...     season = "the spring of class"
```

```
...

>>> Spring.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'Spring' objects>,
'season': 'the spring of class',
'__module__': '__main__',
'__weakref__': <attribute '__weakref__' of 'Spring' objects>,
    '__doc__': None})
```

为了便于观察，将上面的显示结果进行了换行，每个键/值对一行。

从现实的结果中可以发现，有一个键“season”，它是这个类的属性；其值就是类属性的数据。

```
>>> Spring.__dict__['season']
'the spring of class'
>>> Spring.season
'the spring of class'
```

Spring._dict_['season']意思是访问类属性，这是看到上述结果为字典类型而想到的；另外一个我们熟悉的方式就是通过点号，也一样能够实现同样的效果。

下面将这个类实例化，再看看它的实例属性：

```
>>> s = Spring()
>>> s.__dict__
{}
```

实例属性的_dict_是空的。有点奇怪？不奇怪，接着看：

```
>>> s.season
'the spring of class'
```

s.season应该是指向了类属性中的Spring.season，至此，我们其实还没有建立任何实例属性。下面就建立一个实例属性：

```
>>> s.season = "the spring of instance"
>>> s.__dict__
{'season': 'the spring of instance'}
```

这样，实例属性里面就不空了。这时候建立的实例属性和上面的

那个s.season重名，并且把原来的“遮盖”了。这句好是不是熟悉？因为在讲述“实例属性”和“类属性”的时候就提到了，现在读者肯定理解更深入了。

```
>>> s.__dict__['season']
'the spring of instance'
>>> s.season
'the spring of instance'
```

此时，那个类属性如何？我们看看：

```
>>> Spring.__dict__['season']
'the spring of class'
>>> Spring.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'Spring' objects>
>>> Spring.season
'the spring of class'
```

Spring的类属性没有受到实例属性的影响。

按照前面讲述的类属性和实例属性的操作，如果将实例属性（s.season）删除，会不会回到实例属性s._dict_为空呢？

```
>>> del s.season
>>> s.__dict__
{}
>>> s.season
'the spring of class'
```

果然打回原型。

当然，你可以定义其他名称的实例属性，它一样被存储到_dict_里面：

```
>>> s.lang = "python"
>>> s.__dict__
{'lang': 'python'}
>>> s.__dict__['lang']
'python'
```

诚然，这样做仅仅是更改了实例的_dict_内容，对Spring._dict_无任何影响，也就是说通过Spring.lang或者Spring._dict_['lang']是得不到上述结果的。

```
>>> Spring.lang
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Spring' has no attribute 'lang'
>>> Spring.__dict__['lang']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'lang'
```

那么，如果这样操作，会怎样呢？

```
>>> Spring.flower = "peach"
>>> Spring.__dict__
dict_proxy({'__module__': '__main__',
'flower': 'peach',
'season': 'the spring of class',
'__dict__': <attribute '__dict__' of 'Spring' objects>, '__weakref
>>> Spring.__dict__['flower']
'peach'
```

类的_dict_被更改了，类属性中增加了一个flower属性。但是，实例的_dict_中如何？

```
>>> s.__dict__
{'lang': 'python'}
```

没有被修改。然而，还能这样：

```
>>> s.flower
'peach'
```

这个读者是否能解释？其实又回到了前面第一个出现s.season上面了。

通过上面的探讨，是不是基本理解了实例和类的_dict_，并且也看到了属性的变化特点。特别是，这些属性都是可以动态变化的，即你可以随时修改和增删。

属性如此，方法呢？下面就看看方法（类中的函数）。

```
>>> class Spring(object):
...     def tree(self, x):
...         self.x = x
```

```
...          return self.x
...
>>> Spring.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'Spring' objects>,
'__weakref__': <attribute '__weakref__' of 'Spring' objects>,
'__module__': '__main__',
'tree': <function tree at 0xb748fdf4>,
'__doc__': None})

>>> Spring.__dict__['tree']
<function tree at 0xb748fdf4>
```

结果跟前面讨论属性差不多，方法tree()也在__dict__里面。

```
>>> t = Spring()
>>> t.__dict__
{}
```

又跟前面一样。虽然建立了实例，但是在实例的__dict__中没有方法。接下来执行：

```
>>> t.tree("xiangzhangshu")
'xiangzhangshu'
```

还记得前面某章某节有一幅阐述“数据流转”的图吗，其中显示非常明确，当用上面的方式执行方法的时候，实例t与self建立了对应关系，两者是一个外一个内。在方法中self.x＝x，将x的值给了self.x，也就是实例应该拥有这么一个属性。

```
>>> t.__dict__
{'x': 'xiangzhangshu'}
```

果然如此。这也印证了实例t和self的关系，即实例方法（t.tree（'xiangzhangshu'））的第一个参数（self，但没有写出来）绑定实例t，透过self.x来设定值，给t.__dict__添加属性值。

换一个角度再看看：

```
>>> class Spring(object):
...     def tree(self, x):
...         return x
...
```

这个方法中没有将x赋值给self的属性，而是直接return，结果是：

```
>>> s = Spring()
>>> s.tree("liushu")
'liushu'
>>> s.__dict__
{}
```

是不是理解更深入了？

现在需要对Python中的一个观点："一切皆对象"再深入领悟。以上不管是类还是实例的属性和方法，都符合object.attribute格式，并且属性类似。

当你看到这里的时候，要么明白了类和实例的_dict_的特点，要么就糊涂了。糊涂也不要紧，再将上面的重复一遍，特别要自己敲一敲有关代码。

需要说明，我们对_dict_的探讨还留有一个尾巴——属性搜索路径。这个留在后面讲述。

不管是类还是实例，其属性都能随意增加，有时这不是一件好事情，或许在某些时候你不希望别人增加属性。有办法吗？当然有，请继续学习。

4.7.2 _slots_

_slots_能够限制属性的定义，但是这不是它存在的终极目标，它存在的终极目标应该是在编程中非常重要的一个方面：优化内存使用。在某些编程中，优化内存是非常重要的，万万不可忽视。

```
>>> class Spring(object):
...     __slots__ = ("tree", "flower")
...
>>> dir(Spring)
['__class__', '__delattr__', '__doc__', '__format__', '__getattrib
```

仔细看看dir()的结果，还有_dict_属性吗？没有了。也就是说_slots_把_dict_挤出去了，返回来看看，没有_slots_，现在它进入

了类的属性。

```
>>> Spring.__slots__
('tree', 'flower')
```

从这里可以看出，类Spring有且仅有两个属性，并且返回的是一个元组对象。

```
>>> t = Spring()
>>> t.__slots__
('tree', 'flower')
```

实例化之后，实例的_slots_与类的完全一样，这跟前面的_dict_大不一样了。

```
>>> Spring.tree = "liushu"
```

通过类，先赋予一个属性值。然后检验一下实例能否修改这个属性：

```
>>> t.tree = "guangyulan"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Spring' object attribute 'tree' is read-only
```

看来，我们的意图不能达成，报错信息中显示tree这个属性是只读的，不能修改。

```
>>> t.tree
'liushu'
```

因为前面已经通过类给这个属性赋值了，不能用实例属性来修改。只能：

```
>>> Spring.tree = "guangyulan"
>>> t.tree
'guangyulan'
```

用类属性修改。但是对于没有用类属性赋值的，可以通过实例属性：

```
>>> t.flower = "haitanghua"
>>> t.flower
'haitanghua'
```

此时：

```
>>> Spring.flower
<member 'flower' of 'Spring' objects>
```

实例属性的值并没有传回到类属性，你也可以理解为新建立了一个同名的实例属性。如果再给类属性赋值，那么就会这样了：

```
>>> Spring.flower = "ziteng"
>>> t.flower
'ziteng'
```

当然，此时再给t.flower重新赋值，就会报出跟前面一样的错误。

```
>>> t.water = "green"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Spring' object has no attribute 'water'
```

这里试图给实例新增一个属性，也失败了。

看来_slots_已经把实例属性牢牢地管控了起来，但更本质的是优化了内存。诚然，这种优化会在有大量的实例时显出效果。

4.7.3 _getattr_、_setattr_和其他类似方法

结合4.7.2节内容，看一个例子：

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
```

x不是实例的成员（“成员”笼统指类的属性和方法），用a.x访问一定会报错，这是大家所共知的，错误提示中报告了原因：“'A'object has no attribute'x'”

也就是说，如果访问a.x，它不存在，那么就要转向到某个操作。我们把这种情况称之为“拦截”。在Python中，方法就具有这种“拦截”能力。

- _setattr_（self，name，value）：如果要给name赋值，就调用这个方法。
- _getattr_（self，name）：如果name被访问，同时它不存在，此方法被调用。
- _getattribute_（self，name）：当name被访问时自动被调用（注意：这个仅能用于新式类），无论name是否存在，都要被调用。
- _delattr_（self，name）：如果要删除name，这个方法就被调用。

下面用例子说明。

```
>>> class A(object):
...     def __getattr__(self, name):
...         print "You use getattr"
...     def __setattr__(self, name, value):
...         print "You use setattr"
...         self.__dict__[name] = value
```

类A是新式类，除了两个方法，没有别的属性。

```
>>> a = A()
>>> a.x
You use getattr
```

依然调用了不存在的属性a.x，按照开头的例子是要报错的。但是，由于在这里使用了_getattr_（self，name）方法，当发现x不存在于对象的_dict_中时，就调用了_getattr_“拦截成员”。

```
>>> a.x = 7
You use setattr
```

给对象的属性赋值时，调用了_setattr_（self，name，value）方法，这个方法中有一句self._dict_[name]＝value，通过这个语

句，就将属性和数据保存到了对象的_dict_中，如果再调用这个属性：

```
>>> a.x
7
```

x已经存在于对象的_dict_之中。

在上面的类中，当然可以使用_getattribute_（self，name），因为它是新式类，并且，只要访问属性就会调用它。例如：

```
>>> class B(object):
...     def __getattribute__(self, name):
...         print "you are useing getattribute"
...         return object.__getattribute__(self, name)
...
```

为了与前面的类区分，重新搞一个类，在类的方法_etattribute_()中使用return object._getattribute_（self，name）。

再来访问一个不存在的属性：

```
>>> b = B()
>>> b.y
you are useing getattribute
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __getattribute__
AttributeError: 'B' object has no attribute 'y'
>>> b.two
you are useing getattribute
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __getattribute__
AttributeError: 'B' object has no attribute 'two'
```

访问不存在的成员，立刻被_getattribute_拦截了，虽然最后还是要报错的。

```
>>> b.y = 8
>>> b.y
you are useing getattribute
8
```

当给其赋值后，意味着其已经在_dict_里面了，再调用，依然被拦截，但是由于其已经在_dict_内，所以会把结果返回。

特别注意，在这个方法中，没有使用return self._dict_[name]，因为如果用这样的方式就是访问self._dict_，只要访问类的某个属性，就要调用_getattribute_，这样就会导致无限递归下去（死循环）。要避免之。

当你看到这里，是不是觉得上面的方法有点魔力呢？在理解前述内容的基础上，再认真阅读下面的代码，会体会到在实践中的应用。

```
#!/usr/bin/env python
# coding=utf-8
"""
    study __getattr__ and __setattr__
"""

class Rectangle(object):
    """
      the width and length of Rectangle
    """
    def __init__(self):
        self.width = 0
        self.length = 0
    def setSize(self, size):
        self.width, self.length = size
    def getSize(self):
        return self.width, self.length

if __name__ == "__main__":
    r = Rectangle()
    r.width = 3
    r.length = 4
    print r.getSize()
    r.setSize( (30, 40) )
    print r.width
    print r.length
```

上面的代码来自《Beginning Python:From Novice to Professional，Second Editiion》（by Magnus Lie Hetland），根据本教程的需要，稍有修改。

```
$ python 21301.py
(3, 4)
30
40
```

这段代码已经可以正确运行了，但是，作为一个精益求精的程序员，总觉得那种调用方式还有可以改进的空间。比如，要给长宽赋值的时候，必须赋予一个元组，里面包含长和宽。这个能不能改进一下呢？

```
#!/usr/bin/env python
# coding=utf-8
class Rectangle(object):
    def __init__(self):
        self.width = 0
        self.length = 0
    def setSize(self, size):
        self.width, self.length = size
    def getSize(self):
        return self.width, self.length

    size = property(getSize, setSize)

if __name__ == "__main__":
    r = Rectangle()
    r.width = 3
    r.length = 4
    print r.size
    r.size = 30, 40
    print r.width
    print r.length
```

以上代码中因为加了一句size＝property（getSize，setSize），使得调用方法更优雅了。原来用r.getSize()，现在使用r.size，就好像调用一个属性一样。

虽然优化了上面的代码，但是还没有和本节讲述的特殊方法拉上关系，所以，还要继续改写。

```
#!/usr/bin/env python
# coding=utf-8

class NewRectangle(object):
    def __init__(self):
        self.width = 0
        self.length = 0
    def __setattr__(self, name, value):
        if name == "size":
            self.width, self.length = value
        else:
            self.__dict__[name] = value
```

```
    def __getattr__(self, name):
        if name == "size":
            return self.width, self.length
        else:
            raise AttributeError

if __name__ == "__main__":
    r = NewRectangle()
    r.width = 3
    r.length = 4
    print r.size
    r.size = 30, 40
    print r.width
    print r.length
```

除了类的样式变化之外，调用样式没有变，结果是一样的。

4.7.4 获得属性顺序

通过实例获取其属性，如果在_dict_中有，就直接返回其结果；如果没有，会到类属性中找。比如：

```
#!/usr/bin/env python
# coding=utf-8

class A(object):
    author = "qiwsir"
    def __getattr__(self, name):
        if name != "author":
            return "from starter to master."

if __name__ == "__main__":
    a = A()
    print a.author
    print a.lang
```

运行程序：

```
$ python 21302.py
qiwsir
from starter to master.
```

当a＝A()后，并没有为实例建立任何属性，或者说实例的_dict_

是空的（意思是说没有某些属性值）。但是如果要查看a.author，因为实例的属性中没有，所以就去类属性中找，发现果然有，于是返回其值qiwsir。但是，找a.lang时候，不仅实例属性中没有，类属性中也没有，于是就调用了__getattr__()方法。幸好在这个类中有这个方法，如果没有__getattr__()方法呢？如果没有定义这个方法，就会引发AttributeError。

这就是通过实例查找特性的顺序。

4.8 迭代器

迭代对于读者已经不陌生了，已经多次看到这个词语并对其有了初步解了。

我们已经知道，对序列（列表、元组）、字典和文件都可以用iter()方法生成迭代对象，然后用next()方法访问。当然，这种访问不是自动的，如果用for循环，就可以自动完成上述访问了。

如果用dir（list）、dir（tuple）、dir（file）、dir（dict）来查看不同类型对象的属性，会发现它们都有一个名为_iter_的东西。这应该引起读者的关注，因为它和迭代器（iterator）、内置的函数iter()在名字上很像，除了前后的双下画线。望文生义，我们也能猜出它肯定是跟迭代有关的东西。当然，这种猜测也不是没有根据的，其重要根据就是英文单词，如果它们之间没有一点关系，肯定不会将命名设置的一样。

是的，_iter_就是对象的一个特殊方法，它是迭代规则（iterator potocol）的基础。或者说，如果对象没有它，就不能返回迭代器，就没有next()方法，就不能迭代。

如果读者用的是Python 3.x，迭代器对象实现的是_next_()方法，不是next()。并且，在Python 3.x中有一个内建函数next()，可以实现next（it），访问迭代器，这相当于Python 2.x中的it.next()（it是迭代对象）。

4.8.1 _iter_()

类型是list、tuple、file、dict的对象有_iter_()方法，标志着它们能够迭代。这些类型都是Python中固有的，我们能不能自己写一个对象，让它能够迭代呢？

当然可以。

```
#!/usr/bin/env python
# coding=utf-8
```

```
class MyRange(object):
    def __init__(self, n):
        self.i = 0
        self.n = n
    def __iter__(self):
        return self
    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()

if __name__ == "__main__":
    x = MyRange(7)
    print "x.next()==>", x.next()
    print "x.next()==>", x.next()
    print "------for loop--------"
    for i in x:
    print i
```

将代码保存并运行，结果是：

```
$ python 21401.py
x.next()==> 0
x.next()==> 1
------for loop--------
2
3
4
5
6
```

以上代码的含义，是自己仿写了拥有range()的对象，这个对象是可迭代的，分析如下。

（1）_iter_()是类中的核心，它返回了迭代器本身，实现了_iter_()方法的对象，即意味着其可迭代。

（2）含有next()的对象就是迭代器，并且在这个方法中，在没有元素的时候要发起StopIteration()异常。

对以上类的调用换一种方式：

```
if __name__ == "__main__":
    x = MyRange(7)
    print list(x)
    print "x.next()==>", x.next()
```

运行后会出现如下结果：

```
$ python 21401.py
[0, 1, 2, 3, 4, 5, 6]
x.next()==>
Traceback (most recent call last):
  File "21401.py", line 26, in <module>
    print "x.next()==>", x.next()
  File "21401.py", line 21, in next
    raise StopIteration()
StopIteration
```

说明什么呢？print list（x）将对象返回值都装进了列表中并打印出来，这个正常运行了。最终，指针移动到了迭代对象的最后一个，next()方法没有检测，也不知道是不是要停止了，它还要继续下去，当继续下一个的时候，才发现没有元素了，于是返回了StopIteration()。

为什么要用这种可迭代的对象呢？就像上面的例子一样，列表不是挺好的吗？

列表的确非常好，在很多时候效率很高，并且能够解决很多普遍的问题。但是，不要忘记，在某些时候，列表可能会给你带来灾难。因为在你使用列表的时候，需要将列表内容一次性都读入到内存中，这样就增加了内存的负担。如果列表太大，就有内存溢出的危险了，这时候就需要迭代对象。比如斐波那契数列：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Fibs:
    def __init__(self, max):
        self.max = max
        self.a = 0
        self.b = 1

    def __iter__(self):
        return self
```

```
    def next(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
    self.a, self.b = self.b, self.a + self.b
    return fib

if __name__ == "__main__":
    fibs = Fibs(5)
    print list(fibs)
```

运行结果是：

```
$ python 21402.py
[0, 1, 1, 2, 3, 5]
```

给读者一个思考问题：要在斐波那契数列中找出大于1000的最小的数，能不能在上述代码的基础上改造得出呢？

4.8.2 range()和xrange()

关于列表和迭代器之间的区别还有两个非常典型的内建函数：range()和xrange()，研究一下这两个内建函数的差异，会有所收获的。

```
range(...)
    range(stop) -> list of integers
    range(start, stop[, step]) -> list of integers
>>> dir(range)
     ['__call__', '__class__', '__cmp__', '__delattr__', '__doc__'
```

从range()的帮助文档和方法中可以看出，它的结果是一个列表。但是，如果用help（xrange）查看：

```
class xrange(object)
    |  xrange(stop) -> xrange object
    |  xrange(start, stop[, step]) -> xrange object
    |
    |  Like range(), but instead of returning a list, returns an o
    |  generates the numbers in the range on demand.  For looping,
    |  slightly faster than range() and more memory efficient.
```

xrange()返回的是对象，类似range()，但不是列表。在循环的时候，它跟range()相比“slightly faster than range()and more memory efficient”（稍快并具有更高的内存效率）。查看它的方法：

```
>>> dir(xrange)
['__class__', '__delattr__', '__doc__', '__format__', '__getattrib
```

看到令人兴奋的_iter_了吗？说明它是可迭代的，它返回的是一个可迭代的对象。

也就是说，通过range()得到的列表会一次性被读入内存，而xrange()返回的对象，则需要一个数值才返回一个数值。看一个把zip()牵扯进来的例子。

```
>>> zip(range(4), xrange(100000000))
[(0, 0), (1, 1), (2, 2), (3, 3)]
```

第一个range（4）产生的列表被读入内存；第二个很长，但是不用担心，它根本不会产生那么长的列表，因为只需要前4个数值，它就提供前4个数值。如果你要修改为range（100000000），就要花费时间了，但可以尝试一下。

迭代器的确有迷人之处，但是它也不是万能之物。比如迭代器不能回退，只能如过河的卒子，不断向前。另外，迭代器也不适合在多线程环境中对可变集合使用。

4.9 生成器

生成器（generator）是一个非常迷人的东西，也常被认为是Python的高级编程技能。我很乐意在这里跟读者探讨这个话题，因为我相信读者看本书的目的绝非仅仅将自己限制于初学者水平，一定有一颗不羁的心——要成为Python高手。于是乎，需要了解生成器。

“迭代器”已经是很熟悉了吧？生成器和迭代器有着一定的渊源。首先生成器必须是可迭代的，但它又不完全等同于迭代器。

4.9.1 简单的生成器

```
>>> my_generator = (x*x for x in range(4))
```

这是不是跟列表解析很类似呢？仔细观察，它不是列表，像下面这样得到的才是列表：

```
>>> my_list = [x*x for x in range(4)]
```

以上两者的区别在于前者是方括号“[]”后者是圆括号“()”，虽然是细小的差别，但是结果完全不一样。

```
>>> dir(my_generator)
['__class__', '__delattr__', '__doc__', '__format__', '__getattrib
'__iter__',
'__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'next',
'send', 'throw']
```

为了容易观察，将原来得到的结果进行了重新排版。是不是发现了在迭代器中必有的方法_inter_()和next()？这说明my_generator是可迭代的，可以用for循环来依次读出其值。

```
>>> for i in my_generator:
...     print i
...
```

```
0
1
4
9
>>> for i in my_generator:
...     print i
...
```

当第一遍循环的时候，将my_generator里面的值依次读出并打印，但是，若再读一次，就发现没有任何结果（游标已经移动到最后了），这种特性也正是迭代器所具有的。

如果对那个列表，就不一样了：

```
>>> for i in my_list:
...     print i
...
0
1
4
9
>>> for i in my_list:
...     print i
...
0
1
4
9
```

难道生成器就是把列表解析中的“[]”换成“()”这么简单吗？这仅仅是生成器的一种表现形式和基本使用方法罢了，仿照列表解析式的命名，可以称之为“生成器解析式”（或者：生成器推导式、生成器表达式）。

生成器解析式有很多用途，在不少地方可以替代列表解析，特别是针对大数据的时候，Python处理列表时，将全部数据都读入到内存，而迭代器（生成器是迭代器）的优势就在于只将所需要的读入内存里，因此生成器解析式比列表解析式少占内存，再看实例：

```
>>> sum(i*i for i in range(10))
285
```

这个例子是计算1到10以内的自然数的平方和，请观察sum()运算，这样做是不是感觉很迷人？如果是列表，你不得不：

```
>>> sum([i*i for i in range(10)])
285
```

虽然生成器解析式貌似不错，但是对其真正的含义，还需要我们做深入探究才能揭晓。

4.9.2 定义和执行过程

yield这个词在汉语中有“生产、出产”之意，在Python中，它作为一个关键词（变量、函数、类的名称中不能用它来命名），是生成器的标志。

```
>>> def g():
...     yield 0
...     yield 1
...     yield 2
...
>>> g
<function g at 0xb71f3b8c>
```

建立了一个非常简单的函数，跟以往看到的函数唯一不同的地方是用了三个yield语句。然后进行下面的操作：

```
>>> ge = g()
>>> ge
<generator object g at 0xb7200edc>
>>> type(ge)
<type 'generator'>
```

上面建立的函数返回值是一个生成器（generator）类型的对象。

```
>>> dir(ge)
['__class__', '__delattr__', '__doc__', '__format__', '__getattrib
```

在这里看到了__iter__()和next()，说明它是迭代器。既然如此，当然可以：

```
>>> ge.next()
0
>>> ge.next()
1
```

```
>>> ge.next()
2
>>> ge.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

从例子中可以看出，含有yield关键词的函数是一个生成器类型的对象，这个生成器对象是可迭代的。

我们把含有yield语句的函数称作生成器，生成器是一种用普通函数语法定义的迭代器。

通过上面的例子可以看出，这个生成器在定义过程中并没有显化地使用_inter_()和next()，而是只要用了yield语句（yield关键词发起的语句），那个普通函数就神奇般地成为了生成器，也就具备了迭代器的功能特性。

yield语句的作用就是在调用的时候返回相应的值。下面详细剖析一下上面的运行过程。

- ge＝g()：除了返回生成器之外，什么也没有操作，任何值也没有被返回。
- ge.next()：直到这时候，生成器才开始执行，遇到了第一个yield语句，将值返回，并暂停执行（有的称之为挂起）。
- ge.next()：从上次暂停的位置开始，继续向下执行，遇到yield语句，将值返回，又暂停。
- gen.next()：含义与ge.next()相同。
- gene.next()：从上面的挂起位置开始，但是后面没有可执行的了，于是next()发出异常。

从上面的执行过程中会发现yield除了作为生成器的标志之外，还有一个功能就是返回值。那么它跟return这个返回值有什么区别呢？

4.9.3　yield

为了弄清楚yield和return的区别，我们写两个没有什么用途的函数：

```
>>> def r_return(n):
```

```
...     print "You taked me."
...     while n > 0:
...         print "before return"
...         return n
...         n -= 1
...         print "after return"
...
>>> rr = r_return(3)
You taked me.
before return
>>> rr
3
```

从函数被调用的过程可以清晰看出，从rr＝r_return（3）开始，就执行函数体内容了，当遇到return的时候执行该语句，将值返回，然后就结束函数体内的执行，所以return后面的语句根本没有执行。

如果将return改为yield：

```
>>> def y_yield(n):
...     print "You taked me."
...     while n > 0:
...         print "before yield"
...         yield n
...         n -= 1
...         print "after yield"
...
>>> yy = y_yield(3)        #没有执行函数体内语句
>>> yy.next()              #开始执行
You taked me.
before yield
3                       #遇到yield，返回值，并暂停
>>> yy.next()              #从上次暂停位置开始继续执行
after yield
before yield
2                       #又遇到yield，返回值，并暂停
>>> yy.next()
after yield
before yield
1
>>> yy.next()
after yield             #没有满足条件的值，抛出异常
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

结合注释和前面对执行过程的分析，读者一定能理解yield的特点

了，也深知与return的区别了。

一般的函数，都是止于return。作为生成器的函数，由于有了yield，遇到它则程序挂起，如果在之后还有return，遇到它就直接抛出SoptIteration异常而中止迭代。

斐波那契数列已经是老相识了，不论是循环还是迭代都用它举例过，现在还用它举例，只不过要用上yield。

```
#!/usr/bin/env python
# coding=utf-8

def fibs(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1

if __name__ == "__main__":
f = fibs(10)
    for i in f:
    print i ,
```

运行结果如下：

```
$ python 21501.py
1 1 2 3 5 8 13 21 34 55
```

用生成器方式实现的斐波那契数列是不是跟以前的有所不同了呢？读者可以将本书中已经演示过的斐波那契数列实现做对比，体会各种方法的差异。

至此，已经明确，一个函数中，只要包含了yield语句，它就是生成器，也是迭代器。这种方式显然比前面写迭代器的类要简便多了，但这并不意味着迭代器就被抛弃，是用生成器还是用迭代器要根据具体的使用情景而定。

4.9.4 生成器方法

在Python2.5以后，生成器有了一个新特征，就是在开始运行后

能够为生成器提供新的值。这就好似生成器和“外界”之间进行数据交流。

```
>>> def repeater(n):
...     while True:
...         n = (yield n)
...
>>> r = repeater(4)
>>> r.next()
4
>>> r.send("hello")
'hello'
```

当执行到r.next()的时候，生成器开始执行，在内部遇到了yield n挂起。注意在生成器函数中，n＝（yield n）中的yield n是一个表达式，并将结果赋值给n，虽然不严格要求它必须用圆括号包裹，但是一般情况都这么做，请读者也追随这个习惯。

当执行r.send（"hello"）的时候，原来已经被挂起的生成器（函数）又被唤醒，开始执行n＝（yield n），并将send()方法发送的值返回，这就是在运行后能够为生成器提供值的含义。

如果接下来再执行r.next()会怎样？

```
>>> r.next()
```

什么也没有，其实就是返回了None。按照前面的叙述，这次执行r.next()，由于没有给函数的参数传入任何值，yield返回的就只能是None.

还要注意，send()方法必须在生成器运行后并挂起才能使用，即yield至少被执行一次。如果像下面一样就要报错了。

```
>>> s = repeater(5)
>>> s.send("how")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

承接上面的操作，如果将send()的参数设为None，就会把刚才输入的数值返回。

```
>>> s.send(None)
5
```

此外，还有两个方法：close()和throw()。

- throw（type，value=None，traceback=None）：用于在生成器内部（生成器的当前挂起处或未启动时在定义处）抛出一个异常（在yield表达式中）。
- close()：调用时不用参数，用于关闭生成器。

本节最后一句：你在编程中，当然可以不用生成器。

第5章　错误和异常

对于程序报错已经看到过好多次了，那些错误都可以归类为“错误和异常问题”。本章就要对程序运行中出现的错误和异常进行认真研究，近距离观察它们的特点。

5.1　错误

程序员在编写程序的时候，错误往往是难以避免的，可能是因为语法用错了，也可能是拼写错了，当然还可能有其他莫名其妙的错误，比如冒号写成全角的了等。总之，编程中有相当一部分就是要不停地修正错误。

Python中的常见错误之一是语法错误（syntax errors），也是常见的错误。比如：

```
>>> for i in range(10)
  File "<stdin>", line 1
    for i in range(10)
                     ^
SyntaxError: invalid syntax
```

上面那句话因为缺少冒号“:”（英文半角），导致解释器无法解释，于是报错。这个报错行为是由Python的语法分析器完成的，并且检测到了错误所在文件和行号（File"＜stdin＞"，line 1），还以向上箭头“^”标识错误位置，最后一行显示错误类型。

常见错误之二是在没有语法错误时，会出现逻辑错误。逻辑错误可能会由于不完整或者不合法的输入导致，也可能是无法生成、计算等，或者是其他逻辑问题。

当Python检测到一个错误时，解释器就无法继续执行下去，于是抛出相应的信息，这些信息我们笼统地称之为异常信息。

5.2 异常

看一个异常（让0做分母了，小学生都知道会有异常）：

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

当Python抛出异常的时候，首先“跟踪记录（Traceback）”，还可以给它取一个更优雅的名字“回溯”，然后才显示异常的详细信息，标明异常所在位置（文件、行或某个模块）。最后一行是错误类型以及导致异常的原因。

常见的异常如表5-1所示。

表5-1 常见的异常

异 常	描 述
NameError	尝试访问一个没有申明的变量
ZeroDivisionError	除数为 0
SyntaxError	语法错误
IndexError	索引超出序列范围
KeyError	请求一个不存在的字典关键字
IOError	输入/输出错误（比如你要读的文件不存在）
AttributeError	尝试访问未知的对象属性

为了能够深入理解，依次举例，展示异常的出现条件和结果。

- NameError

```
>>> bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

Python中虽然不需要在使用变量之前先声明类型，但也需要对变量进行赋值，然后才能使用，不被赋值的变量，不能在Python中存在，因为变量相当于一个标签，要把它贴到对象上才有意义。

- ZeroDivisionError

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

你或许有足够信心，貌似这样简单的错误在你的编程中是不会出现的，但在实际情境中，可能没有这么容易识别，所以，依然要小心。

- SyntaxError

```
>>> for i in range(10)
  File "<stdin>", line 1
    for i in range(10)
                     ^
SyntaxError: invalid syntax
```

这种错误发生在Python代码编译的时候，当编译到这一句时，解释器不能将代码转化为Python字节码就报错，它是在程序运行之前出现的。现在有不少编辑器都有语法校验功能，在你写代码的时候就能显示出语法的正误，这多少会对编程者有帮助。

- IndexError

```
>>> a = [1, 2, 3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> d = {"python":"itdiffer.com"}
>>> d["java"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'java'
```

这两个都属于“鸡蛋里面挑骨头”类型，一定得报错了。不过在编程实践中，特别是循环的时候，常常由于循环条件设置不合理出现这种类型的错误。

- IOError

```
>>> f = open("foo")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'foo'
```

如果你确认有文件，就一定要把路径写正确，因为你并没有告诉Python要对你的Computer进行全身搜查。Python只会按照你指定的位置去找，找不到就异常。

- AttributeError

```
>>> class A(object): pass
...
>>> a = A()
>>> a.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'foo'
```

属性不存在，出现错误。

Python内建的异常也不仅仅是上面几个，上面只是列出常见的异常中的几个，还有：

```
>>> range("aaa")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer end argument expected, got str.
```

总之，如果读者在调试程序的时候遇到了异常，不要慌张，这是好事情，是Python在帮助你修改错误。只要认真阅读异常信息，再用dir()、help()或者官方网站文档、Google等来协助，一定能解决问题。

5.3 处理异常

在一段程序中，为了能够让程序健壮，有时还要处理异常。举例：

```
#!/usr/bin/env python
# coding=utf-8

while 1:
    print "this is a division program."
    c = raw_input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = raw_input("first number:")
        b = raw_input("second number:")
        try:
            print float(a)/float(b)
            print "*************************"
        except ZeroDivisionError:
            print "The second number can't be zero!"
            print "*************************"
    else:
        break
```

运行这段程序，显示如下过程：

```
$ python 21601.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:5
second number:2
2.5
*************************
this is a division program.
input 'c' continue, otherwise logout:c
first number:5
second number:0
The second number can't be zero!
*************************
this is a division program.
input 'c' continue, otherwise logout:d
$
```

从运行情况看，当在第二个数，即除数为0时，程序并没有因为这个错误而停止，而是给用户一个友好的提示，让用户有机会改正错误。这完全得益于程序中“处理异常”的设置，如果没有“处理异常”，则当异常出现时就会导致程序中止。

5.3.1 try...except...

对于前述举例程序，只看try和except部分，如果没有异常发生，except子句在try语句执行之后被忽略；如果try子句中有异常发生，该部分的其他语句被忽略，直接跳到except部分，执行其后面指定的异常类型及其子句。

except后面也可以没有任何异常类型，即无异常参数。如果这样，不论try部分发生什么异常，都会执行except。

在except子句中，可以根据异常或者别的需要，进行更多的操作。比如：

```
#!/usr/bin/env python
# coding=utf-8

class Calculator(object):
    is_raise = False
    def calc(self, express):
        try:
            return eval(express)
        except ZeroDivisionError:
            if self.is_raise:
                print "zero can not be division."
            else:
                raise
```

先解释函数eval()，它的含义是：

```
eval(...)
    eval(source[, globals[, locals]]) -> value

    Evaluate the source in the context of globals and locals.
    The source may be a string representing a Python expression
    or a code object as returned by compile().
    The globals must be a dictionary and locals can be any mapping
    defaulting to the current globals and locals.
    If only globals is given, locals defaults to it.
```

例如：

```
>>> eval("3+5")
8
```

另外，在except子句中，还有一个孤零零的raise，作为单独一个语句，它的含义是将异常信息抛出，并且except子句用了一个判断语句，根据不同的情况确定走不同的分支。

```
if __name__ == "__main__":
    c = Calculator()
    print c.calc("8/0")
```

故意出现0做分母的情况，就是要让is_raise＝False，则会：

```
$ python 21602.py
Traceback (most recent call last):
  File "21602.py", line 17, in <module>
    print c.calc("8/0")
  File "21602.py", line 8, in calc
    return eval(express)
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

如果将is_raise的值改为True，会是这样：

```
if __name__ == "__main__":
    c = Calculator()
    c.is_raise = True
    print c.calc("8/0")
```

运行结果：

```
$ python 21602.py
zero can not be division.
None
```

最后的None是c.calc（"8/0"）的返回值，因为有print c.calc（"8/0"），所以被打印出来。

5.3.2　处理多个异常

处理多个异常并不是因为同时报出多个异常，程序在运行中，只要遇到一个异常就会有反应，所以，每次捕获到的异常一定是一个。所谓处理多个异常的意思是可以容许捕获不同的异常，由不同的except子句处理。

```
#!/usr/bin/env python
# coding=utf-8

while 1:
    print "this is a division program."
    c = raw_input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = raw_input("first number:")
        b = raw_input("second number:")
        try:
            print float(a)/float(b)
            print "*************************"
        except ZeroDivisionError:
            print "The second number can't be zero!"
            print "*************************"
        except ValueError:
            print "please input number."
            print "************************"
    else:
        break
```

修改一下程序，增加了一个except子句，目的是当用户输入的不是数字时，捕获并处理这个异常。测试如下：

```
$ python 21701.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:3
second number:"hello"           #输入了一个不是数字的东西
please input number.            #对照上面的程序，捕获并处理了这个异常
************************
this is a division program.
input 'c' continue, otherwise logout:c
first number:4
second number:0
The second number can't be zero!
*************************
this is a division program.
input 'c' continue, otherwise logout:4
```

如果有多个except，try里面遇到一个异常，就转到相应的except

子句，其他的忽略。如果except没有相应的异常，该异常也会抛出，不过这时程序就要中止了，因为异常“浮出”程序顶部。

除了用多个except之外，还可以在一个except后面放多个异常参数，比如上面的程序，可以将except部分修改为：

```
except (ZeroDivisionError, ValueError):
    print "please input rightly."
    print "********************"
```

运行的结果就是：

```
$ python 21701.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:2
second number:0            #捕获异常
please input rightly.
********************
this is a division program.
input 'c' continue, otherwise logout:c
first number:3
second number:a            #异常
please input rightly.
********************
this is a division program.
input 'c' continue, otherwise logout:d
```

需要注意的是，except后面如果是多个参数，一定要用圆括号包裹起来。否则，后果自负。

在对异常的处理中，前面都是自己写一个提示语，但自己写的不如内置的异常错误提示好，如果希望把默认错误提示打印出来，但程序还不能中断，怎么办？Python提供了一种方式，将上面的代码修改如下：

```
while 1:
    print "this is a division program."
    c = raw_input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = raw_input("first number:")
        b = raw_input("second number:")
        try:
            print float(a)/float(b)
            print "*************************"
```

```
        except (ZeroDivisionError, ValueError), e:
            print e
            print "********************"
    else:
        break
```

运行一下，看看提示信息。

```
$ python 21702.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:2
second number:a                         #异常
could not convert string to float: a
********************
this is a division program.
input 'c' continue, otherwise logout:c
first number:2
second number:0                         #异常
float division by zero
********************
this is a division program.
input 'c' continue, otherwise logout:d
```

在Python 3.x中，常常这样写：except（ZeroDivisionError，ValueError）as e:

在上面的程序中，只处理了两个异常，还可能有更多的异常，如果要处理，怎么办？可以这样：execpt:或者except Exception，e，后面什么参数也不写就好了。

5.3.3 else子句

有了try...except...，在一般情况下是够用的，但总有不一般的时候出现，所以，就增加了一个else子句。其实，人类的自然语言何尝不是如此呢？总要根据需要添加不少东西。

```
>>> try:
...     print "I am try"
... except:
...     print "I am except"
... else:
...     print "I am else"
```

```
...
I am try
I am else
```

这段演示能够帮助读者理解else的执行特点。如果执行了try，则except被忽略，但是else被执行。

```
>>> try:
...     print 1/0
... except:
...     print "I am except"
... else:
...     print "I am else"
...
I am except
```

这时候else就不被执行了。

理解了else的执行特点，可以写这样一段程序，还是类似于前面的计算，只是如果输入的有误，就不断要求重新输入，直到输入正确并得到了结果，才不再要求输入内容，然后程序结束。

在看下面的参考代码之前，读者是否可以先自己写一段并调试？看看结果如何。

```
#!/usr/bin/env python
# coding=utf-8
while 1:
    try:
        x = raw_input("the first number:")
        y = raw_input("the second number:")

        r = float(x)/float(y)
        print r
    except Exception, e:
        print e
        print "try again."
    else:
        break
```

先看运行结果：

```
$ python 21703.py
the first number:2
the second number:0          #异常，执行except
```

```
float division by zero
try again.                   #循环
the first number:2
the second number:a          #异常
could not convert string to float: a
try again.
the first number:4
the second number:2          #正常，执行try
2.0                          #然后else：break，退出程序
```

相当满意的执行结果。

程序中的“except Exception，e”的含义是不管什么异常，这里都会捕获，并且传给变量e，然后用print e把异常信息打印出来。

5.3.4 finally子句

finally子句，一听这个名字，就感觉它是做善后工作的。的确如此，如果有了finally，不管前面执行的是try，还是except，最终都要执行它。因此有一种说法是将finally用在可能的异常后进行清理。比如：

```
>>> x = 10

>>> try:
...     x = 1/0
... except Exception, e:
...     print e
... finally:
...     print "del x"
...     del x
...
integer division or modulo by zero
del x
```

看一看x是否被删除？

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

当然，在应用中可以将上面的各个子句都综合起来使用，写成如

下样式：

```
try:
    do something
except:
    do something
else:
    do something
finally
    do something
```

看到这里，你是不是觉得这个“try…except…”跟“if…else…”有点相似呢？但不要误以为某一个就可以取代另外一个，他们的存在都是有道理的。

5.3.5 assert语句

```
>>> assert 1==1
>>> assert 1==0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

从上面的举例中可以基本了解assert的特点。

assert，翻译过来是“断言”之意。assert是语句等价于布尔真的判定，发生异常就意味着表达式为假。

assert的应用情景与其翻译的意思“断言”一样，即当程序运行到某个节点的时候，就断定某个变量的值必然是什么，或者对象必然拥有某个属性等，简单说就是断定什么东西必然是什么，如果不是，就抛出错误。

```
#!/usr/bin/env python
# coding=utf-8

class Account(object):
    def __init__(self, number):
        self.number = number
        self.balance = 0

    def deposit(self, amount):
```

```
        assert amount > 0
        self.balance += balance

    def withdraw(self, amount):
        assert amount > 0
        if amount <= self.balance:
            self.balance -= amount
        else:
            print "balance is not enough."
```

程序中，deposit()和withdraw()方法的参数amount值必须是大于零的，这里就用断言，如果不满足条件就会报错。比如这样来运行：

```
if __name__ == "__main__":
    a = Account(1000)
    a.deposit(-10)
```

出现的结果是：

```
$ python 21801.py
Traceback (most recent call last):
  File "21801.py", line 22, in <module>
    a.deposit(-10)
  File "21801.py", line 10, in deposit
    assert amount > 0
AssertionError
```

这就是断言assert的作用。什么是使用断言的最佳时机？

- 如果没有特别的目的，断言应该用于如下情况：
- 防御性的编程。
- 运行时对程序逻辑的检测。
- 合约性检查（比如前置条件，后置条件）。
- 程序中的常量。
- 检查文档。

（上述要点来自《Python使用断言的最佳时机》网址：http://www.oschina.net/translate/when-to-use-assert）

不论是否理解，都要先看看，请牢记，在具体的开发过程中，有时间就看看本书，不断加深对这些概念的理解，这也是master的成就之法。

最后，引用危机百科中对“异常处理”词条的说明，作为对“错误和

异常”部分的总结（有所删改）：

异常处理是编程语言或计算机硬件里的一种机制，用于处理软件或信息系统中出现的异常状况（即超出程序正常执行流程的某些特殊条件）。

各种编程语言在处理异常方面具有非常显著的不同点（错误检测与异常处理的区别在于：错误检测是在正常的程序流中，处理不可预见问题的代码，例如一个调用操作未能成功结束）。某些编程语言有这样的函数：当输入存在非法数据时不能被安全地调用，或者返回值不能与异常进行有效的区别。例如，C语言中的atoi函数（从ASCII串到整数的转换）在输入非法时可以返回0。在这种情况下编程者需要另外进行错误检测（可能通过某些辅助全局变量，如C的errno），或进行输入检验（如通过正则表达式），或者共同使用这两种方法。

通过异常处理，我们可以对用户在程序中的非法输入进行控制和提示，以防程序崩溃。

从进程的视角来看，硬件中断相当于可恢复异常，虽然中断一般与程序流本身无关。

从子程序编程者的视角来看，异常是很有用的一种机制，用于通知外界该子程序不能正常执行，如输入的数据无效（例如除数是0），或所需资源不可用（例如文件丢失）。如果系统没有异常机制，则编程者需要用返回值来标示发生了哪些错误。

Python语言对异常处理机制是非常普遍深入的，所以想写出不含try、except的程序非常困难。

第6章　模块

随着对Python学习的深入，其优点日渐突出，让读者也感觉到Python的强大了，强大感觉之一就是“模块自信”，因为Python不仅有自带的模块（称之为标准库），还有海量的第三方模块，并且很多开发者还在不断贡献自己开发的新模块，正是有了这么强大的“模块自信”，Python才被很多人钟爱。并且这种方式也正在不断被其他更多语言所借鉴，几乎成为普世行为了（不知道Python是不是首倡者）。

“模块自信”的本质是：开放。

Python不是一个封闭的体系，而是一个开放系统。开放系统的最大好处就是避免了“熵增”。

熵的概念是由德国物理学家克劳修斯于1865年所提出，是一种测量在动力学方面不能做功的能量总数，也就是当总体的熵增加，其做功能力也下降，熵的量度正是能量退化的指标。

熵亦被用于计算一个系统中的失序现象，也就是计算该系统混乱的程度。

根据熵的统计学定义，热力学第二定律说明一个孤立系统倾向于增加混乱程度。换句话说就是对于封闭系统而言，会越来越趋向于无序化。反过来，开放系统则能避免无序化。

6.1 编写模块

想必读者已经熟悉了import语句，曾经有这样一个例子：

```
>>> import math
>>> math.pow(3,2)
9.0
```

这里的math（是Python标准库之一，在本章，我们要逐渐理解模块、库之类的术语。）就是一个模块，用import引入这个模块，然后可以使用模块里面的函数，比如pow()函数。显然，这里是不需要自己动手写具体函数的，我们的任务就是拿过来使用。这就是模块的好处：拿过来就用，不用自己重写。

6.1.1 模块是程序

“模块是程序”一语道破了模块的本质，它就是一个扩展名为.py的Python程序。

我们能够在应该使用它的时候将它引用过来，节省精力，不需要重写雷同的代码。

但是，如果我自己写一个.py文件，是不是就能作为模块import过来呢？还不那么简单。必须得让Python解释器能够找到你写的模块。比如，在某个目录中，我写了这样一个文件：

```
#!/usr/bin/env python
# coding=utf-8
lang = "python"
```

并把它命名为pm.py，那么这个文件就可以作为一个模块被引入。不过由于这个模块是我自己写的，Python解释器并不知道，得先告诉它我写了这样一个文件。

```
>>> import sys
>>> sys.path.append("~/Documents/VBS/StartLearningPython/2code/pm.
```

用这种方式告诉Python解释器，我写的那个文件在哪里。在这个方法中，也用了模块import sys，不过由于sys是Python标准库之一，所以不用特别告诉Python解释器其位置。

上面那个一长串的地址是Ubuntu系统的地址格式，如果读者使用的是Windows系统，请写你所保存的文件路径。

```
>>> import pm
>>> pm.lang
'python'
```

在pm.py文件中有一个赋值语句，即lang＝"python"，现在将pm.py作为模块引入（注意作为模块引入的时候不带扩展名），就可以通过“模块名字”＋“.”＋“属性或方法名称”来访问pm.py中的东西。当然，如果要访问不存在的属性，肯定是要报错的。

```
>>> pm.xx
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'xx'
```

请读者回到pm.py文件的存储目录，查看一下是不是多了一个扩展名是.pyc的文件？

解释器，英文是：interpreter，在Python中，它的作用就是将.py的文件转化为.pyc文件，而.pyc文件是由字节码（bytecode）构成的，然后计算机执行.pyc文件。

很多人喜欢将这个世界简化再简化，比如编程语言就分为解释型和编译型，不但如此，还将两种类型的语言分别贴上运行效率高低的标签，解释型的运行速度就慢，编译型的运行速度就快。一般人都把Python看成是解释型的，于是就得出它运行速度慢的结论。不少人都因此上当受骗了，认为Python不值得学，或者做不了什么“大事”。这就是将本来复杂的、多样化的世界非得划分为“黑白”的结果，喜欢用“非此即彼”的思维方式考虑问题。

世界是复杂的，“敌人的敌人就是朋友”是幼稚的，“一分为二”是机械的。

如同刚才看到的那个.pyc文件一样，当Python解释器读取了.py文件，先将它变成由字节码组成的.pyc文件，然后这个.pyc文件交给一

个叫作Python虚拟机的东西去运行（那些号称编译型的语言也是这个流程，不同的是它们先有一个明显的编译过程，编译好了之后再运行）。如果.py文件修改了，Python解释器会重新编译，只是这个编译过程不全显示给你看。

有了.pyc文件后，每次运行就不需要重新让解释器来编译.py文件了，除非.py文件修改了。这样，Python运行的就是那个编译好了的.pyc文件。

是否还记得前面写有关程序然后执行时常常要用到if_name_ = ="_main_"，那时我们直接用“python filename.py”的格式来运行该文件，此时我们也同样有了.py文件，不过是作为模块引入的。这就得深入探究一下，同样是.py文件，它怎么知道是被当作程序执行还是被当作模块引入？

为了便于比较，将pm.py文件进行改造。

```
#!/usr/bin/env python
# coding=utf-8

def lang():
    return "python"

if __name__ == "__main__":
    print lang()
```

沿用先前的做法：

```
$ python pm.py
python
```

如果将这个程序作为模块，导入，会是这样的：

```
>>> import sys
>>> sys.path.append("~/Documents/VBS/StarterLearningPython/2code/p
>>> import pm
>>> pm.lang()
'python'
```

查看模块属性和方法，可以使用dir()。

```
>>> dir(pm)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 
```

同样一个.py文件，可以把它当作程序来执行，也可以将它作为模块引入。

```
>>> __name__
'__main__'
>>> pm.__name__
'pm'
```

如果要作为程序执行，则_name_=="_main_"；如果作为模块引入，则pm._name_=="pm"，即变量_name_的值是模块名称。

用这种方式就可以区分是执行程序还是作为模块引入了。

在一般情况下，如果仅仅是用作模块引入，不必写if_name_=="_main_"。

6.1.2　模块的位置

为了让我们自己写的模块能够被Python解释器知道，需要用sys.path.append（"~/Documents/VBS/StarterLearningPython/2code/pm.py"）。其实，在Python中，所有模块都被加入到了sys.path里面。用下面的方法可以看到模块所在位置：

```
>>> import sys
>>> import pprint
>>> pprint.pprint(sys.path)
['',
 '/usr/local/lib/python2.7/dist-packages/autopep8-1.1-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/pep8-1.5.7-py2.7.egg',
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-i386-linux-gnu',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages/PILcompat',
 '/usr/lib/python2.7/dist-packages/gtk-2.0',
 '/usr/lib/python2.7/dist-packages/ubuntu-sso-client',
 '~/Documents/VBS/StarterLearningPython/2code/pm.py']
```

从中也发现了我自己写的那个文件。

凡在上面列表所包括位置内的.py文件都可以作为模块引入。不妨举个例子，把前面自己编写的pm.py文件修改为pmlib.py，然后复制到'/usr/lib/python2.7/dist-packages中。（这是以Ubuntu为例说明，如果是其他操作系统，读者用类似方法也能找到。）

```
$ sudo cp pm.py /usr/lib/python2.7/dist-packages/pmlib.py
[sudo] password for qw:

$ ls /usr/lib/python2.7/dist-packages/pm*
/usr/lib/python2.7/dist-packages/pmlib.py
```

文件放到了指定位置。看下面的：

```
>>> import pmlib
>>> pmlib.lang
<function lang at 0xb744372c>
>>> pmlib.lang()
'python'
```

将模块文件放到指定位置是一种不错的方法，但感觉此法受到了拘束，程序员都喜欢自由，能不能放到别处呢？

当然能，用sys.path.append()就是不管把文件放在哪里，都可以把其位置告诉Python解释器。虽然这种方法在前面用了，但其实是很不常用的，因为它也有麻烦的地方，比如在交互模式下，如果关闭了，再开启，还得重新告知。

比较常用的方法是设置PYTHONPATH环境变量。

环境变量，不同的操作系统设置方法略有差异。读者可以根据自己的操作系统，到网上搜索设置方法。

以Ubuntu为例，建立一个Python的目录，然后将我自己写的.py文件放到这里，并设置环境变量。

```
:~$ mkdir python
:~$ cd python
:~/python$ cp ~/Documents/VBS/StarterLearningPython/2code/pm.py my
:~/python$ ls
mypm.py
```

然后将这个目录~/python，即/home/qw/python设置环境变量。

```
vim /etc/profile
```

要用root权限，在打开的文件最后增加export PATH=/home/qw/python:$PAT，然后保存退出即可。

注意，我是在~/python目录下输入Python，然后进入到交互模式：

```
:~$ cd python
:~/python$ python

>>> import mypm
>>> mypm.lang()
'python'
```

如此，就完成了告知过程。

6.1.3 _all_在模块中的作用

上面的模块虽然比较简单，但是已经显示了编写模块，以及在程序中导入模块的基本方式。在实践中，所编写的模块也许更复杂一点，比如，有这么一个模块，其文件命名为pp.py

```
# /usr/bin/env python
# coding:utf-8

public_variable = "Hello, I am a public variable."
_private_variable = "Hi, I am a private variable."

def public_teacher():
    print "I am a public teacher, I am from JP."

def _private_teacher():
    print "I am a private teacher, I am from CN."
```

接下来就是熟悉的操作了，进入到交互模式中。pp.py这个文件就是一个模块，该模块中包含了变量和函数。

```
>>> import sys
>>> sys.path.append("~/Documents/StarterLearningPython/2code/pp.py
>>> import pp
>>> from pp import *
>>> public_variable
'Hello, I am a public variable.'
>>> _private_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_private_variable' is not defined
```

变量public_variable能够被使用，但是另外一个变量_private_variable不能被调用，先观察一下两者的区别，后者是以单下画线开头的，这样的是私有变量。而from pp import*的含义是“希望能访问模块（pp）中有权限访问的全部名称”，那些被视为私有的变量或者函数或者类，当然就没有权限被访问了。

再如：

```
>>> public_teacher()
  I am a public teacher, I am from JP.
>>> _private_teacher()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_private_teacher' is not defined
```

这不是绝对的，但如果要访问具有私有性质的东西，可以这样做。

```
>>> import pp
>>> pp._private_teacher()
I am a private teacher, I am from CN.
>>> pp._private_variable
'Hi, I am a private variable.'
```

下面再对pp.py文件进行改写，增加一些东西。

```
# /usr/bin/env python
# coding:utf-8

__all__ = ['_private_variable', 'public_teacher']

public_variable = "Hello, I am a public variable."
_private_variable = "Hi, I am a private variable."
```

```
def public_teacher():
    print "I am a public teacher, I am from JP."

def _private_teacher():
        print "I am a private teacher, I am from CN."
```

在修改之后的pp.py中，增加了_all_变量以及相应的值，在列表中包含了一个私有变量的名字和一个函数的名字。这是在告诉引用本模块的解释器，这两个东西是有权限被访问的，而且只有这两个东西。

```
>>> import sys
>>> sys.path.append("~/Documents/StarterLearningPython/2code/pp.py
>>> from pp import *
>>> _private_variable
'Hi, I am a private variable.'
```

果然，曾经不能被访问的私有变量，现在能够访问了。

```
>>> public_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'public_variable' is not defined
```

因为这个变量没有在_all_的值中，虽然以前曾经被访问到过，但是现在就不行了。

```
>>> public_teacher()
I am a public teacher, I am from JP.
>>> _private_teacher()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_private_teacher' is not defined
```

这只不过是再次说明前面的结论罢了。当然，如果以import pp引入模块，再用pp._private_teacher的方式是一样有效的。

6.1.4 包和库

顾名思义，包和库都是比“模块”大的。一般来讲，一个“包”里面会有多个模块，当然，“库”是一个更大的概念了，比如Python标准库

中的每个库都有好多个包，每个包都有若干个模块。

一个包由多个模块组成，即有多个.py的文件，那么这个所谓的“包”就是我们熟悉的一个目录罢了。现在需要解决如何引用某个目录中的模块问题。解决方法就是在该目录中放一个_init_.py文件。_init_.py是一个空文件，将它放在某个目录中，就可以将该目录中的其他.py文件作为模块被引用。

例如，建立一个目录，名曰：package_qi，里面依次放了pm.py和pp.py两个文件，然后建立一个空文件_init_.py

接下来，需要导入这个包（package_qi）中的模块。

下面这种方法很清晰明了。

```
>>> import package_qi.pm
>>> package_qi.pm.lang()
'python'
```

下面这种方法，貌似简短，但如果多了，恐怕难以分辨。

```
>>> from package_qi import pm
>>> pm.lang()
'python'
```

在后续制作网站的实战中，还会经常用到这种方式，届时会了解更多。请保持兴趣继续阅读，不要半途而废，不然疑惑得不到解决，好东西就看不到了。

6.2 自带电池

“Python自带‘电池’”，听说过这种说法吗？

在Python被安装的时候，就有不少模块也随着安装到本地的计算机上了。这些东西就如同“能源”、“电力”一样，让Python拥有了无限生机，能够非常轻而易举地免费使用很多模块。所以，称之为“自带电池”。

那些在安装Python时就默认已经安装好的模块被称为“标准库”。

熟悉标准库是编程之必须。

6.2.1 引用方式

所有模块都服从下述引用方式，是最基本的、也是最常用的，还是可读性非常好的：

```
import modulename
```

例如：

```
>>> import pprint
>>> a = {"lang":"python", "book":"www.itdiffer.com", "teacher":"qi
>>> pprint.pprint(a)
{'book': 'www.itdiffer.com',
 'goal': 'from beginner to master',
 'lang': 'python',
 'teacher': 'qiwsir'}
```

在对模块进行说明的过程中，以标准库pprint为例。

以pprint.pprint()的方式使用模块中的一种方法，这种方法能够让字典格式化输出。看看结果是不是比原来更容易阅读了呢？

在import后面，理论上可以跟好多模块名称，但是在实践中，还是建议大家一次跟一个名称，太多了看着头晕眼花，不容易阅读。

这是用import pprint的样式引入，并以点号“.”（英文半角）的形式引用其方法。

还有下面的方式：

```
>>> from pprint import pprint
```

意思是从pprint模块中只将pprint()引入，之后就可以直接使用它了。

```
>>> pprint(a)
{'book': 'www.itdiffer.com',
 'goal': 'from beginner to master',
 'lang': 'python',
 'teacher': 'qiwsir'}
```

再懒惰一些，可以：

```
>>> from pprint import *
```

将pprint模块中的一切都引入了，于是可以像上面那样直接使用每个函数。但是，这样造成的结果是可读性不是很好，并且，不管是不是用得到，都拿过来，是不是太贪婪了？贪婪的结果是内存会消耗不少。所以，这种方法可以用于常用的并且模块属性或方法不是很多的情况，莫贪婪。

诚然，如果很明确使用模块中的哪些方法或属性，那么使用类似from modulename import name1，name2，name3...也未尝不可。需要再次提醒的是不能因为引入了模块而降低了可读性，让别人不知道呈现在眼前的方法是从何而来。

有时候引入的模块或者方法名称有点长，可以给它重命名。如：

```
>>> import pprint as pr
>>> pr.pprint(a)
{'book': 'www.itdiffer.com',
 'goal': 'from beginner to master',
 'lang': 'python',
 'teacher': 'qiwsir'}
```

当然，还可以这样：

```
>>> from pprint import pprint as pt
>>> pt(a)
{'book': 'www.itdiffer.com',
 'goal': 'from beginner to master',
 'lang': 'python',
 'teacher': 'qiwsir'}
```

但是不管怎么样，一定要让人看懂，且要过了若干时间，自己也还能看懂。记住：软件很多时候是给人看的，只是偶尔让机器执行。

6.2.2 深入探究

继续以pprint为例，深入研究：

```
>>> import pprint
>>> dir(pprint)
['PrettyPrinter', '_StringIO', '__all__', '__builtins__', '__doc__
```

对dir()并不陌生。从结果中可以看到pprint的属性和方法。其中有不少是以双画线、单画线开头的。但为了不影响我们的视觉，先把它们去掉。

```
>>> [ m for m in dir(pprint) if not m.startswith('_') ]
['PrettyPrinter', 'isreadable', 'isrecursive', 'pformat', 'pprint'
```

针对这几个，为了能够搞清楚它们的含义，可以使用help()，比如：

```
>>> help(isreadable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'isreadable' is not defined
```

这样做是错误的，知道错在何处吗？

```
>>> help(pprint.isreadable)
```

前面是用import pprint方式引入模块的：

```
Help on function isreadable in module pprint:
```

```
isreadable(object)
    Determine if saferepr(object) is readable by eval().
```

通过帮助信息，能够查看到该方法的详细说明。可以用这种方法一个一个地查过来，对每个方法都熟悉一些。

注意pprint.PrettyPrinter是一个类，后面的是函数（方法）。

再回头看看dir（pprint）的结果：

```
>>> pprint.__all__
['pprint', 'pformat', 'isreadable', 'isrecursive', 'saferepr', 'Pr
```

这个结果是不是眼熟？除了"warnings"，跟前面通过列表解析式得到的结果一样。

其实，当我们使用from pprint import*的时候，就是将_all_里面的方法引入，如果没有这个，就会将其他所有属性、方法等引入，包括那些以双画线或者单画线开头的变量、函数，事实上这些东西很少在引入模块时被使用。

6.2.3 帮助、文档和源码

你能记住每个模块的属性和方面吗？比如前面刚刚查询过的pprint模块中的属性和方法，现在能背诵出来吗？我的记忆力不行，都记不住。所以，我非常喜欢使用dir()和help()，这也是本书从开始到现在，乃至到以后，总在提倡的方式。

```
>>> print pprint.__doc__
Support to pretty-print lists, tuples, & dictionaries recursively.

Very simple, but useful, especially in debugging data structures.

Classes
-------

PrettyPrinter()
    Handle pretty-printing operations onto a stream using a config
    set of formatting parameters.

Functions
---------
```

```
pformat()
    Format a Python object into a pretty-printed representation.

pprint()
    Pretty-print a Python object to a stream [default is sys.stdou

saferepr()
    Generate a 'standard' repr()-like value, but protect against r
    data structures.
```

pprint._doc_是查看整个类的文档，还知道整个文档写在什么地方吗？

还是使用pm.py那个文件，增加如下内容：

```
#!/usr/bin/env python
# coding=utf-8

"""                                                #增加的
This is a document of the python module.           #增加的
"""                                                #增加的

def lang():
    ...                                            #省略了，后面的也省略了
```

在这个文件的开始部分，在所有类、方法和import之前，写一个用三个引号包括的字符串，这就是文档。

```
>>> import sys
>>> sys.path.append("~/Documents/VBS/StarterLearningPython/2code")
>>> import pm
>>> print pm.__doc__

This is a document of the python module.
```

这就是撰写模块文档的方法，即在.py文件的最开始写相应的内容，这个要求应该成为开发者的习惯。

对于Python的标准库和第三方模块，不仅可以看帮助信息和文档，还能够查看源码，因为它是开放的。

还是回头到dir（pprint）中找一找，有一个_file_，它就告诉我们这个模块的位置：

```
>>> print pprint.__file__
/usr/lib/python2.7/pprint.pyc
```

我是在Ubuntu中操作，读者要注意观察自己的操作系统结果。

虽然是.pyc文件，但是不用担心，根据显示的目录，找到相应的.py文件即可。

```
$ ls /usr/lib/python2.7/pp*
/usr/lib/python2.7/pprint.py  /usr/lib/python2.7/pprint.pyc
```

果然有一个pprint.py，打开它，就看到源码了。

```
$ cat /usr/lib/python2.7/pprint.py

...

"""Support to pretty-print lists, tuples, & dictionaries recursive

Very simple, but useful, especially in debugging data structures.

Classes
-------

PrettyPrinter()
    Handle pretty-printing operations onto a stream using a config
    set of formatting parameters.

Functions
---------

pformat()
    Format a Python object into a pretty-printed representation.

....
"""
```

我只查抄了文档中的部分信息，是不是跟前面通过_doc_查看的结果一样呢？

请读者在闲暇时间阅读源码。事实证明，这种标准库中的源码是质量最好的。阅读高质量的代码，是提高编程水平的途径之一。

6.3 标准库

Python标准库的内容非常多，有人专门为此写过一本书。在本书中，我将根据自己的理解和喜好，选几个呈现出来，一来显示标准库之强大功能，二来演示如何理解和使用标准库。

6.3.1 sys

这是一个跟Python解释器关系密切的标准库，前面已经使用过sys.path.append()。

```
>>> import sys
>>> print sys.__doc__
```

显示了sys的基本文档，第一句话概括了本模块的基本特点。

```
This module provides access to some objects used or maintained by
interpreter and to functions that interact strongly with the inter
```

在诸多sys函数和变量中，选择常用的来说明。

1.sys.argv

sys.argv是变量，专门用来向Python解释器传递参数，所以名曰“命令行参数”。

先解释什么是命令行参数。

```
$ python --version
Python 2.7.6
```

这里的--version就是命令行参数，如果你使用python--help可以看到更多：

```
$ python --help
```

```
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-B     : don't write .py[co] files on import; also PYTHONDONTWRITE
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prom
         if stdin does not appear to be a terminal; also PYTHONINS
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE
-OO    : remove doc-strings in addition to the -O optimizations
-R     : use a pseudo-random salt to make hash() values of various
         unpredictable between separate invocations of the interpr
         a defense against denial-of-service attacks
```

只选择了部分内容摆在这里。所看到的如-B、-h之流，都是参数，比如python-h，其功能同上，那么-h也是命令行参数。

sys.arg在Python中的作用就是这样，通过它可以向解释器传递命令行参数。比如：

```
#!/usr/bin/env python
# coding=utf-8

import sys

print "The file name: ", sys.argv[0]
print "The number of argument", len(sys.argv)
print "The argument is: ", str(sys.argv)
```

将上述代码保存，文件名是22101.py。然后如此做：

```
$ python 22101.py
The file name:  22101.py
The number of argument 1
The argument is:  ['22101.py']
```

将结果和前面的代码做个对照。

（1）在$python 22101.py中，“22101.py”是要运行的文件名，同时也是命令行参数，是前面的Python这个指令的参数，其地位与python-h中的参数-h是等同的。

（2）sys.argv[0]是第一个参数，就是上面提到的22101.py，即

文件名。

如果这样来试试：

```
$ python 22101.py beginner master www.itdiffer.com
The file name:  22101.py
The number of argument 4
The argument is:  ['22101.py', 'beginner', 'master', 'www.itdiffer
```

在这里用sys.arg[1]得到的就是beginner，依次类推。

2.sys.exit()

这个方法的意思是退出当前程序。

```
Help on built-in function exit in module sys:

exit(...)
    exit([status])

    Exit the interpreter by raising SystemExit(status).
    If the status is omitted or None, it defaults to zero (i.e., s
    If the status is an integer, it will be used as the system exi
    If it is another kind of object, it will be printed and the sy
    exit status will be one (i.e., failure).
```

从文档信息中可知，如果用sys.exit()退出程序，会返回SystemExit异常。这里先告知读者，还有另外一种退出方式：os._exit()，这两个有所区别。

```
#!/usr/bin/env python
# coding=utf-8

import sys

for i in range(10):
    if i == 5:
        sys.exit()
    else:
        print i
```

这段程序的运行结果就是：

```
```

```
$ python 22102.py
0
1
2
3
4
```

在有的函数中（甚至大多数函数中）会用到return，其含义是终止当前的函数，并返回相应值（如果没有就是None）。但是sys.exit()的含义是退出当前程序，并发起SystemExit异常。这就是两者的区别了。

使用sys.exit（0）表示正常退出，如果退出的时候有一个对人友好的提示信息，可以用sys.exit（"I wet out at here."），那么字符串信息就被打印出来。

3.sys.path

sys.path已经不陌生了，它可以查找模块所在的目录，以列表的形式显示出来。如果用append()方法，就能够向这个列表增加新的模块目录。

4.sys.stdin，sys.stdout，sys.stderr

将这三个放到一起，是因为他们的变量都是类文件流对象，分别表示标准UNIX概念中的标准输入、标准输出和标准错误。与Python功能对照，sys.stdin获得输入（用raw_input()输入的通过它获得，Python 3.x中是imput()），sys.stdout负责输出。

流是程序输入或输出的一个连续的字节序列，设备（例如鼠标、键盘、磁盘、屏幕、调制解调器和打印机）的输入和输出都是用流来处理的。程序在任何时候都可以使用它们。一般来讲，stdin（输入）并不一定来自键盘，stdout（输出）也并不一定显示在屏幕上，它们都可以重定向到磁盘文件或其他设备上。

还记得print()吧，它的本质就是sys.stdout.write（object＋'\n'）。

```
sys.stdout.write(object + '\n')。
>>> for i in range(3):
...     print i
```

```
...
0
1
2

>>> import sys
>>> for i in range(3):
...     sys.stdout.write(str(i))
```

造成上面的输出结果在表象上有如此差异的，原因就是那个'\n'的有无。

```
>>> for i in range(3):
...     sys.stdout.write(str(i) + '\n')
...
0
1
2
```

从这里可以看出，两者是完全等效的。如果仅仅止于此，则意义不大。更强大的在于通过sys.stdout能够做到将输出内容从“控制台”转到“文件”，称之为重定向。这样也许控制台看不到（很多时候这个不重要），但是文件中已经有了要输出的内容。

```
>>> f = open("stdout.md", "w")
>>> sys.stdout = f
>>> print "Learn Python: From Beginner to Master"
>>> f.close()
```

当sys.stdout＝f之后，就意味着将输出目的地转到了打开（建立）的文件中，然后用print，将内容“打印”到那个文件中，在控制台就不显现。

打开文件看看便知：

```
$ cat stdout.md
Learn Python: From Beginner to Master
```

这就是标准输出。

另外两个，输入和错误也类似，读者可以自行测试。

6.3.2 copy

前面对浅拷贝和深拷贝做了研究，这里再次提出，即是复习，也是凑数，以显得我考虑到了这个常用模块。

```
>>> import copy
>>> copy.__all__
['Error', 'copy', 'deepcopy']
```

这个模块中常用的就是copy和deepcopy。

为了具体说明，看这样一个例子：

```
#!/usr/bin/env python
# coding=utf-8

import copy

class MyCopy(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return str(self.value)

foo = MyCopy(7)

a = ["foo", foo]
b = a[:]
c = list(a)
d = copy.copy(a)
e = copy.deepcopy(a)

a.append("abc")
foo.value = 17

print "original: %r\n slice: %r\n list(): %r\n copy(): %r\n deepco
```

保存并运行：

```
$ python 22103.py
original: ['foo', 17, 'abc']
 slice: ['foo', 17]
 list(): ['foo', 17]
 copy(): ['foo', 17]
```

```
deepcopy(): ['foo', 7]
```

读者可以对照结果和程序，就能理解各种拷贝的实现方法和含义了——深拷贝和浅拷贝。

6.3.3 os

os模块提供了访问操作系统服务的功能，它所包含的内容比较多，有时候感觉很神秘。

```
>>> import os
>>> dir(os)
['EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST
```

这么多内容不能都介绍，列出来纯粹是要吓唬你一下，先混个脸熟，将来用到哪个了，可以到这里来找。

下面介绍的都是我自认为用得比较多的，如果读者要用某个方法或属性，但是这里没有介绍，你完全可以自己用help()来自学，当然，还有另外一个好工具——Google（内事不决问Google，外事不明问谷歌）。

1.操作文件：重命名、删除文件

在对文件进行操作的时候，open()这个内建函数可以建立、打开文件。但是，如果对文件进行改名、删除操作，就要使用os模块的方法了。

首先建立一个文件，文件名为22201.py，文件内容是：

```
#!/usr/bin/env python
# coding=utf-8

print "This is a tmp file."
```

然后将这个文件名称修改为别的名称。

```
>>> import os
>>> os.rename("22201.py", "newtemp.py")
```

注意，我是先进入到了文件22201.py的目录，然后再进入交互模式，所以，可以直接写文件名，如果不是这样，需要将文件的路径写上。

os.rename（"22201.py"，"newtemp.py"）中，第一个文件是原文件名称，第二个是打算修改成为的文件名。

然后查看，能够看到这个文件。

```
$ ls new*
newtemp.py
```

文件内容可以用cat newtemp.py查看（这是在Ubuntu系统，如果是Windows系统，可以用其相应的编辑器打开文件看内容）。

除了修改文件名称，还可以修改目录名称，请注意阅读帮助信息。

```
Help on built-in function rename in module posix:

rename(...)
    rename(old, new)

    Rename a file or directory.
```

另外一个os.remove()，首先看帮助信息，然后再实验。

```
Help on built-in function remove in module posix:

remove(...)
    remove(path)

    Remove a file (same as unlink(path)).
```

为了测试，先建立一些文件。

```
$ pwd
/home/qw/Documents/VBS/StarterLearningPython/2code/rd
```

这是我建立的临时目录，里面有几个文件：

```
$ ls
```

```
a.py  b.py  c.py
```

下面删除a.py文件。

```
>>> import os
>>> os.remove("/home/qw/Documents/VBS/StarterLearningPython/2code/
```

看看删了吗？

```
$ ls
b.py  c.py
```

果然管用，再来一个狠的：

```
>>> os.remove("/home/qw/Documents/VBS/StarterLearningPython/2code/
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 21] Is a directory: '/home/qw/Documents/VBS/Starte
```

报错了。我打算将这个目录下的所剩文件删光，但这么做不行。注意帮助中的那句Remove a file，os.remove()就是用来删除文件的。并且从报错中也可以看到，错误的原因在于那个参数是一个目录。

要删除目录，还得继续向下学习。

2.操作目录

（1）os.listdir：显示目录中的文件。

```
Help on built-in function listdir in module posix:

listdir(...)
    listdir(path) -> list_of_strings

Return a list containing the names of the entries in the directory

    path: path of directory to list

The list is in arbitrary order.  It does not include the special
entries '.' and '..' even if they are present in the directory.
```

看完帮助信息，读者一定觉得这是一个非常简单的方法，不过，

要特别注意它返回的值是列表，且不显示文件夹中用特殊格式命名的文件（它们是隐藏文件）。在Linux中，用ls命令也看不到这些隐藏的文件。

```
>>> os.listdir("/home/qw/Documents/VBS/StarterLearningPython/2code
['b.py', 'c.py']
>>> files = os.listdir("/home/qw/Documents/VBS/StarterLearningPyth
>>> for f in files:
...     print f
...
b.py
c.py
```

（2）os.getcwd，os.chdir：当前工作目录，改变当前工作目录。

这两个函数怎么用？唯有通过help()看文档了，请读者自行看看，就不贴出来了，仅演示一个例子：

```
>>> cwd = os.getcwd()          #当前目录
>>> print cwd
/home/qw/Documents/VBS/StarterLearningPython/2code/rd
>>> os.chdir(os.pardir)        #进入到上一级

>>> os.getcwd()                #当前
'/home/qw/Documents/VBS/StarterLearningPython/2code'

>>> os.chdir("rd")             #进入下级

>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
```

os.pardir的功能是获得父级目录，相当于“..”。

```
>>> os.pardir
'..'
```

（3）os.makedirs，os.removedirs：创建和删除目录。

直接上例子：

```
>>> dir = os.getcwd()
>>> dir
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
```

```
>>> os.removedirs(dir)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/os.py", line 170, in removedirs
    rmdir(name)
OSError: [Errno 39] Directory not empty:
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
```

什么时候都不能得意忘形，一定要谦卑，从看文档开始一点一点地理解。看报错信息，要删除某个目录，则那个目录必须是空的。

```
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
```

这是当前目录，在这个目录下再建一个新的子目录：

```
>>> os.makedirs("newrd")
>>> os.chdir("newrd")
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code/newrd'
```

建立了一个。下面把刚刚建立的这个目录删除，毫无疑问它是空的。

```
>>> os.listdir(os.getcwd())
[]
>>> newdir = os.getcwd()
>>> os.removedirs(newdir)
```

按照我的理解，这里应该报错。因为我是在当前工作目录删除当前工作目录，如果这样能够执行，总觉得有点别扭。但事实上行得通，就算是Python的规定吧。不过，若让我来确定这个功能的话，还是习惯不能在本地删除本地。

按照上面的操作，再看当前的工作目录：

```
>>> os.getcwd()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory
```

目录被删了，只能回到父级。

```
>>> os.chdir(os.pardir)
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
```

有点不可思议，本来没有当前工作目录，怎么会有“父级”呢？但现实就是这样。

补充一点，前面说的如果目录不空，就不能用os.removedirs()删除。但是，可以用模块shutil的retree方法。

```
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
>>> os.chdir("rd")
>>> now = os.getcwd()
>>> now
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
>>> os.listdir(now)
['b.py', 'c.py']
>>> import shutil
>>> shutil.rmtree(now)
>>> os.getcwd()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory
```

请读者注意，对于os.makedirs()还有这样的特点：

```
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
>>> d0 = os.getcwd()
>>> d1 = d0+"/ndir1/ndir2/ndir3"    #这是想建立的目录，但是ndir1,ndir2
>>> d1
'/home/qw/Documents/VBS/StarterLearningPython/2code/ndir1/ndir2/nd
>>> os.makedirs(d1)
>>> os.chdir(d1)
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code/ndir1/ndir2/nd
```

不存在的目录也被建立起来，直到做右边的目录为止。与os.makedirs()类似的还有os.mkdir()，不过，os.mkdir()没有上面这个功能，它只能一层一层地建目录。os.removedirs()和os.rmdir()也类似，区别也类似上面。

3.文件和目录属性

不管是哪种操作系统，都能看到文件或者目录的有关属性，那么，在os模块中，也有这样的一个方法：os.stat()。

```
>>> p = os.getcwd()     #当前目录
>>> p
'/home/qw/Documents/VBS/StarterLearningPython'
```

显示这个目录的有关信息：

```
>>> os.stat(p)
posix.stat_result(st_mode=16895, st_ino=4L, st_dev=26L, st_nlink=1
```

再指定一个文件：

```
>>> pf = p + "/README.md"
```

显示此文件的信息：

```
>>> os.stat(pf)
posix.stat_result(st_mode=33279, st_ino=67L, st_dev=26L, st_nlink=
```

从结果中看，可能看不出什么来，先不用着急。这样的结果对computer姑娘是友好的，但可能对读者不友好。如果用下面的方法，就友好多了：

```
>>> fi = os.stat(pf)
>>> mt = fi[8]
```

fi[8]就是st_mtime的值，它代表最后modified（修改）文件的时间。看结果：

```
>>> mt
1429580969
```

还是不友好，下面用time模块来友好一下：

```
>>> import time
>>> time.ctime(mt)
'Tue Apr 21 09:49:29 2015'
```

现在就对读者友好了。

用os.stat()能够查看文件或者目录的属性。如果要修改呢？比如在部署网站的时候，常常要修改目录或者文件的权限等。这种操作在Python的os模块能做到吗？

要求越来越多了。一般情况下，不在Python里做这个，当然，世界是复杂的，肯定有人会用到的，所以os模块提供了os.chmod()

4.操作命令

读者如果使用某种Linux系统，或者曾经用过DOS（恐怕很少），或者在Windows里面用过command，对敲命令都不陌生。通过命令来做事情的确是很酷的，比如，我在Ubuntu中，要查看文件和目录，只需要ls就足够了。我并不是否认图形界面，对于某些人（比如程序员）在某些情况下，命令是不错的选项，甚至是离不开的。

os模块中提供了这样的方法，许可程序员在Python程序中使用操作系统的命令。（以下是在Ubuntu系统，如果读者是Windows系统，可以将命令换成DOS命令。）

```
>>> p
'/home/qw/Documents/VBS/StarterLearningPython'
>>> command = "ls " + p
>>> command
'ls /home/qw/Documents/VBS/StarterLearningPython'
```

为了输入方便，采用了前面例子中已经有的那个目录，并且，用拼接字符串的方式，将要输入的命令（查看某文件夹下的内容）组装成一个字符串，赋值给变量command，然后：

```
>>> os.system(command)
01.md     101.md  105.md  109.md  113.md  117.md  121.md  125.md  1
02.md     102.md  106.md  110.md  114.md  118.md  122.md  126.md  1
03.md     103.md  107.md  111.md  115.md  119.md  123.md  127.md  1
0images   104.md  108.md  112.md  116.md  120.md  124.md  128.md  1
0
```

这样就列出来了该目录下的所有内容。

需要注意的是，os.system()是在当前进程中执行命令，直到它执行结束。如果需要一个新的进程，可以使用os.exec或者os.execvp。

对此有兴趣详细了解的读者，可以查看帮助文档了解。另外，os.system()通过shell执行命令，执行结束后将控制权返回到原来的进程，但是os.exec()及相关的函数，则在执行后不将控制权返回到原继承，从而使Python失去控制。

关于Python对进程的管理，此处暂不介绍。

os.system()是一个用途很多的函数。曾有一个朋友网上询问，用它来启动浏览器。不过，这个操作的确要非常仔细，为什么呢？演示一下就明白了。

```
>>> os.system("/usr/bin/firefox")

(process:4002): GLib-CRITICAL **: g_slice_set_config: assertion 's

(firefox:4002): GLib-GObject-WARNING **: Attempt to add property G
......
```

我是在Ubuntu上操作的，浏览器的地址是/usr/bin/firefox，可是，若朋友是Windows系统，那么就要非常小心了，因为在Windows里面，表示路径的斜杠跟上面显示的是反着的，可是在Python中“\”代表转义。比较简单的一个方法是用r"c:\user\firfox.exe"的样式，因为在r""中的，都被认为是原始字符。而且在Windows系统中，一般情况下那个文件不是安装在我演示的那个简单样式的文件夹中，而是安装在“C:\Program Files”，这中间有空格，所以还要注意空格问题。读者按照这些提示，看看能不能完成用os.system()启动firefox的操作。

凡是感觉麻烦的东西，必然有另外简单的方法来替代。于是又有了一个webbrowser模块，可以专门用来打开指定网页。

```
>>> import webbrowser
>>> webbrowser.open("http://www.itdiffer.com")
True
```

不管是什么操作系统，只要如上操作就能打开网页。

真是神奇的标准库，有如此多的工具，能不加速开发进程吗？能不降低开发成本吗？“人生苦短，我用Python”！

6.3.4 heapq

堆（heap），是一种数据结构，引用维基百科中的说明：

堆（英语：heap），是计算机科学中一类特殊的数据结构的统称。堆通常是一个可以被看作一棵树的数组对象。

对于这个新的概念，读者不要心慌意乱或者恐惧，因为它本质上不是新东西，而是在我们已经熟知的知识基础上扩展出来的内容。

堆的实现是通过构造二叉堆，也就是一种二叉树。

1.基本知识

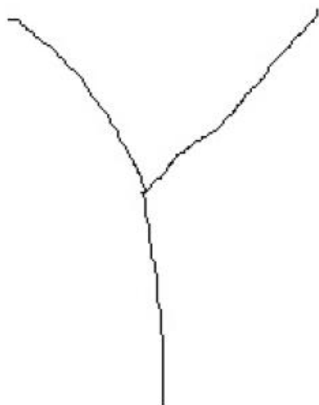
这是一颗在苏州很常见的香樟树，马路两边、公园里随处可见，特别是在艳阳高照的时候，它的树荫能把路面遮盖。



但是，在编程中，我们常说的树是这样的：

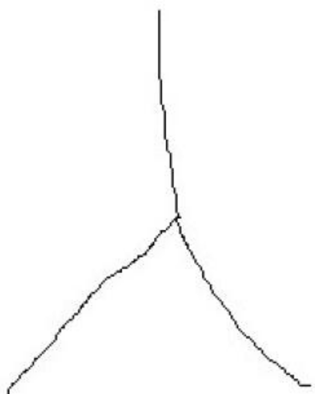


这是一棵“根”在上面的树，也是编程中常说的树。为什么会这样呢？我想主要是画着更方便吧。上面那棵树虽然根在上面了，还完全是写实的作品，但本人作为一名隐姓埋名多年的抽象派画家，不喜欢这样的树，我画出来是这样的：

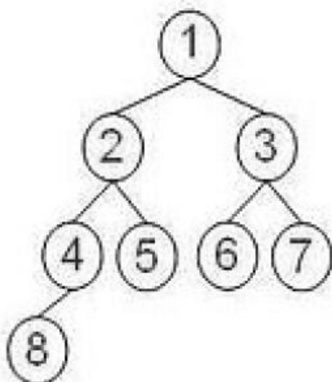


这棵树有两根枝杈，可不要小看这两根枝杈，《道德经》上说“一生二，二生三，三生万物”。一就是下面那个树干，二就是两个枝杈，每个枝杈还可以看作下一个一，然后再有两个枝杈，如此不断重复（这简直就是递归呀），就成为了一棵大树。

这棵树画成这样就更符合编程的习惯了，可以向下不断延伸。



并且给它一个正规的名字：二叉树。

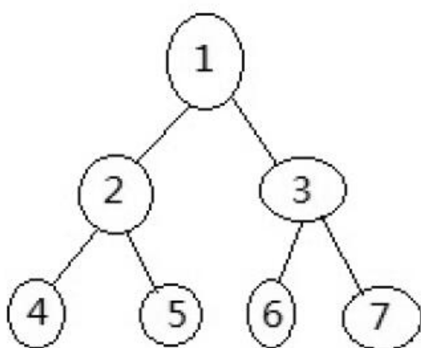


这个也是二叉树，完全脱胎于我所画的后现代抽象主义作品，但也略有不同，这幅图在各个枝杈上显示的是数字。这种类型的“树”就是编程语言中所说的二叉树，维基百科曰：

在计算机科学中，二叉树（英语：Binary tree）是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。

在上图的二叉树中，最顶端的那个数字相当于树根，也称作“根”。每个数字所在位置成为一个节点，每个节点向下分散出两个“子节点”。并不是所有节点都有两个子节点，比如上图的最后一层，这类二叉树又称为完全二叉树（Complete Binary Tree），有的二叉树，所有的节点都有两个子节点，这类二叉树称作满二叉树

（Full Binarry Tree），如下图。



下面讨论的对象是通过二叉树实现的，其具有如下特点：

- 节点的值大于等于（或者小于等于）任何子节点的值。
- 节点左子树和右子树是一个二叉堆。如果父节点的值总大于等于任何一个子节点的值，其为最大堆；若父节点的值总小于等于子节点值，则为最小堆。上面图示中的完全二叉树，就表示一个最小堆。

堆的类型还有别的，如斐波那契堆等，但很少用。所以，通常将二叉堆也说成堆。下面所说的堆就是二叉堆，而二叉堆又是用二叉树实现的。

2.堆的存储

堆用列表来表示，如图6-1所示。

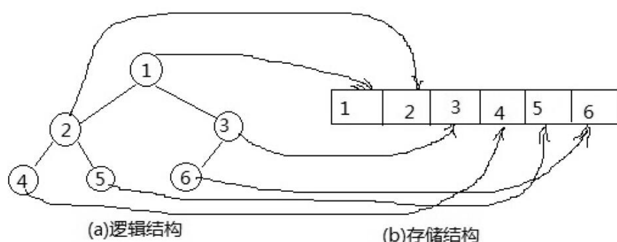
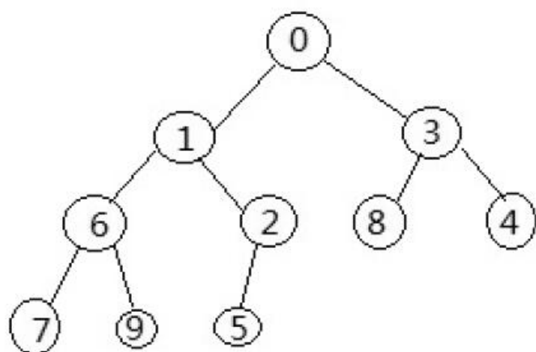


图6-1　用列表来表示堆

从图示中可以看出，将逻辑结构中的树的节点数字依次填入到存

储结构中。看这个图，似乎是列表中按照顺序进行排列似的。但是，这仅仅是由于那个树的特点造成的，如果是下面的树：



将上面的逻辑结构转换为存储结构，读者就能看出来了，不再是按照顺序排列的了。

关于堆的各种操作，如插入、删除、排序等，本节不会专门叙述，读者可以参阅有关资料。下面要介绍如何用Python中的模块heapq来实现这些操作。

3.heapq模块

heapq中的heap是堆，q就是queue（队列）的缩写。此模块包括：

```
>>> import heapq
>>> heapq.__all__
['heappush', 'heappop', 'heapify', 'heapreplace', 'merge', 'nlarge
```

依次查看这些函数的使用方法。

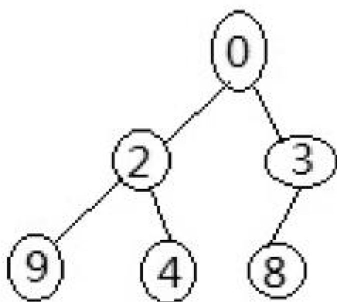
- heappush（heap，x）：将x压入堆heap（这是一个列表）。

```
Help on built-in function heappush in module _heapq:

heappush(...)
    heappush(heap, item) -> None. Push item onto heap, maintaining
```

```
>>> import heapq
>>> heap = []
>>> heapq.heappush(heap, 3)
>>> heapq.heappush(heap, 9)
>>> heapq.heappush(heap, 2)
>>> heapq.heappush(heap, 4)
>>> heapq.heappush(heap, 0)
>>> heapq.heappush(heap, 8)
>>> heap
[0, 2, 3, 9, 4, 8]
```

请读者注意上面的操作，在向堆增加数值的时候并没有严格按照什么顺序，是随意的。但是，当查看堆的数据时，显示的是一个有一定顺序的数据结构。这种顺序不是按照从小到大，而是按照前面所说的完全二叉树的方式排列，显示的是存储结构，可以把它还原为逻辑结构，看看是不是一棵二叉树。



由此可知，利用heappush()函数将数据放到堆里面之后，会自动按照二叉树的结构进行存储。

- heappop（heap）：删除最小元素。

承接上面的操作：

```
>>> heapq.heappop(heap)
0
>>> heap
[2, 4, 3, 9, 8]
```

用heappop()函数，从heap堆中删除了一个最小元素，并且返回该值。但是，这时候的heap显示顺序，并非简单地将0去除，而是按照完全二叉树的规范重新进行排列。

- heapify()：将列表转换为堆。

如果已经建立了一个列表，利用heapify()可以将列表直接转化为堆。

```
>>> hl = [2, 4, 6, 8, 9, 0, 1, 5, 3]
>>> heapq.heapify(hl)
>>> hl
[0, 3, 1, 4, 9, 6, 2, 5, 8]
```

经过这样的操作，列表hl就变成了堆（堆的顺序和列表不同），可以对hl（堆）使用heappop()或者heappush()等函数了。否则，不可。

```
>>> heapq.heappop(hl)
0
>>> heapq.heappop(hl)
1
>>> hl
[2, 3, 5, 4, 9, 6, 8]
>>> heapq.heappush(hl, 9)
>>> hl
[2, 3, 5, 4, 9, 6, 8, 9]
```

不要认为堆里面只能放数字，举例中之所以用数字，是因为对它的逻辑结构比较好理解。

```
>>> heapq.heappush(hl, "q")
>>> hl
[2, 3, 5, 4, 9, 6, 8, 9, 'q']
>>> heapq.heappush(hl, "w")
>>> hl
[2, 3, 5, 4, 9, 6, 8, 9, 'q', 'w']
```

- heapreplace()。

是heappop()和heappush()的联合，也就是删除一个的同时再加入一个。例如：

```
>>> heap
[2, 4, 3, 9, 8]
>>> heapq.heapreplace(heap, 3.14)
2
>>> heap
```

```
[3, 4, 3.14, 9, 8]
```

先简单罗列关于堆的几个常用函数。那么堆在编程实践中的用途有哪些呢？排序是一个应用方面。一提到排序，读者肯定想到的是sorted()或者列表中的sort()，这两个都是常用的函数，而且在一般情况下已经足够使用了。但如果使用堆排序，相对于其他排序，也有自己的优势。不同的排序方法有不同的特点，读者可以自行深入研究不同排序的优劣。

6.3.5 deque

有这样一个问题：一个列表，比如是[1，2，3]，在最右边增加一个数字。

这也太简单了，不就是用append()这个内建函数追加一个吗？

这是简单，但能不能在最左边增加一个数字呢？

这个应该有办法，不过得想想。读者在向下阅读之前，能不能想出一个方法来？

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
>>> nl = [7]
>>> nl.extend(lst)
>>> nl
[7, 1, 2, 3, 4]
```

你或许还有别的方法。但是，Python为我们提供了一个更简单的模块来解决这个问题。

```
>>> from collections import deque
```

这里用这种引用方法是因为collections模块中东西很多，我们只用到deque。

```
>>> lst
[1, 2, 3, 4]
```

还是这个列表，试试分别从右边和左边增加数字。

```
>>> qlst = deque(lst)
```

这是必需的，将列表转化为deque。deque在汉语中有一个名字，叫作“双端队列”。（double-ended queue）。

```
>>> qlst.append(5)          #从右边增加
>>> qlst
deque([1, 2, 3, 4, 5])
>>> qlst.appendleft(7)      #从左边增加
>>> qlst
deque([7, 1, 2, 3, 4, 5])
```

这样操作多么容易呀，继续看删除：

```
>>> qlst.pop()
5
>>> qlst
deque([7, 1, 2, 3, 4])
>>> qlst.popleft()
7
>>> qlst
deque([1, 2, 3, 4])
```

删除也分左右。下面这个，请读者仔细观察。

```
>>> qlst.rotate(3)
>>> qlst
deque([2, 3, 4, 1])
```

rotate()的功能是将[1，2，3，4]的首位连起来，你就想象一个圆环，在上面有1、2、3、4几个数字。如果一开始正对着你的是1，依顺时针方向排列，就是从1开始的数列，如图6-2所示。

经过rotate()，这个环就发生旋转了，如果是rotate（3），表示每个数字按照顺时针方向前进三个位置，于是变成了如图6-3所示的样子。

请原谅我的后现代主义超级抽象派作图方式。从图中可以看出，数列变成了[2，3，4，1]。rotate()就好像在拨转这个圆环。

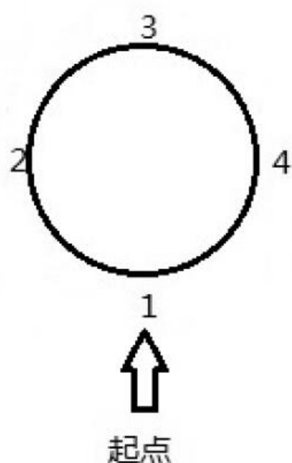


图6-2　数字排列

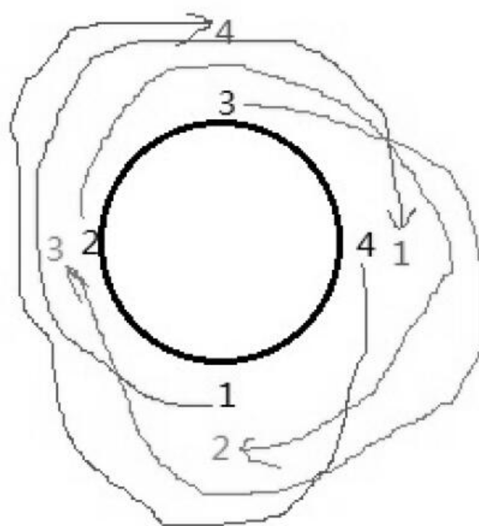


图6-3　数字按顺时针方向前进

请原谅我的后现代主义超级抽象派作图方式。从图中可以看出，数列变成了[2，3，4，1]。rotate()就好像在拨转这个圆环。

```
>>> qlst
deque([3, 4, 1, 2])
>>> qlst.rotate(-1)
```

```
>>> qlst
deque([4, 1, 2, 3])
```

如果参数是负数，那么就逆时针转。

在deque中，还有extend和extendleft方法，读者可自己调试。

6.3.6 calendar

```
>>> import calendar
>>> cal = calendar.month(2015, 1)
>>> print cal
    January 2015
Mo Tu We Th Fr Sa Su
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

轻而易举得到了2015年1月的日历，并且排列还那么整齐，这就是calendar模块。读者可以用dir()去查看这个模块下的所有内容。为了让读者阅读方便，将常用的整理如下：

- calendar(year,w=2,l=1,c=6)

返回year年年历，3个月一行，间隔距离为c，每日宽度间隔为w字符，每行长度为21*W+18+2*C，l是每星期行数。

```
>>> year = calendar.calendar(2015)
>>> print year
```

因为显示的内容太多，所以只将部分内容截取下来，如图6-4所示。

```
                                  2015

      January                   February                   March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
          1  2  3  4                         1                         1
 5  6  7  8  9 10 11       2  3  4  5  6  7  8       2  3  4  5  6  7  8
12 13 14 15 16 17 18       9 10 11 12 13 14 15       9 10 11 12 13 14 15
19 20 21 22 23 24 25      16 17 18 19 20 21 22      16 17 18 19 20 21 22
26 27 28 29 30 31         23 24 25 26 27 28         23 24 25 26 27 28 29
                                                    30 31
```

图6-4　2015年日历

其他部分就是按照上面的样式，将2015年度的各个月份日历完全显示出来。

- isleap(year)

判断是否为闰年，是则返回True，否则False。

```
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2015)
False
```

怎么判断一年是闰年的问题，常常见诸一些编程语言的练习题，现在用一个方法搞定。

- leapdays(y1,y2)

返回在y1、y2两年之间的闰年总数，包括y1，但不包括y2，这有点如同序列的切片。

```
>>> calendar.leapdays(2000,2004)
1
>>> calendar.leapdays(2000,2003)
1
```

- month(year, month, w=2, l=1)

返回year年month月日历，两行标题，一周一行。每日宽度间隔为w字符，每行的长度为7*w+6，l是每星期的行数。

```
>>> print calendar.month(2015, 5)
      May 2015
Mo Tu We Th Fr Sa Su
```

```
                1   2   3
 4   5   6   7   8   9  10
11  12  13  14  15  16  17
18  19  20  21  22  23  24
25  26  27  28  29  30  31
```

- monthcalendar(year, month)

返回一个列表，列表内的元素还是列表，这叫作嵌套列表。每个子列表代表一个星期，都是从星期一到星期日，如果没有本月的日期，则为0。

```
>>> calendar.monthcalendar(2015, 5)
[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10], [11, 12, 13, 14, 1
```

读者可以将这个结果和calendar.month（2015，5）去对照理解。

- monthrange(year,month)

返回一个元组，里面有两个整数。第一个整数代表着该月的第一天从星期几开始（从0开始，依次为星期一、星期二……6代表星期日）。第二个整数代表该月一共多少天。

```
>>> calendar.monthrange(2015, 5)
(4, 31)
```

从返回值可知，2015年5月1日是星期五，这个月一共31天。这个结果，也可以从日历中看到。

- weekday(year,month,day)

输入年月日，知道该日是星期几（注意，返回值依然按照从0到6依次对应星期一到星期六）。

```
>>> calendar.weekday(2015, 5, 4)     #星期一
0
>>> calendar.weekday(2015, 6, 4)     #星期四
3
```

6.3.7 time

time模块很常用，比如记录某个程序运行时间长短等，下面一一道来其中的方法。

- time()

```
>>> import time
>>> time.time()
1430745298.391026
```

time.time()获得的是当前时间（严格说是时间戳），只不过这个时间对人不友好，它是以1970年1月1日0时0分0秒为计时起点，到当前的时间长度（不考虑闰秒）。

UNIX时间，或称POSIX时间是UNIX或类UNIX系统使用的时间表示方式：从格林威治时间1970年1月1日0时0分0秒起至现在的总秒数，不考虑闰秒。

现时大部分使用的UNIX的系统都是32位的，即它们会以32位二进制数字表示时间。但是它们最多只能表示至协调世界时间2038年1月19日3时14分07秒（二进制：01111111111111111111111111111111，0x7FFF:FFFF），在下一秒二进制数字会是10000000000000000000000000000000，（0x8000:0000），这是负数，因此各系统会把时间误解作1901年12月13日20时45分52秒（亦有说回归到1970年）。这时可能会令软件发生问题，导致系统瘫痪。

目前的解决方案是把系统由32位转为64位。在64位系统下，此时间最多可以表示到292，277，026，596年12月4日15时30分08秒。

有没有对人友好一点的时间显示呢？

- localtime()

```
>>> time.localtime()
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=4, tm_hour=21, tm
```

这个就友好多了。得到的结果可以称之为时间元组（也有括号），其各项的含义如表6-1所示。

表6-1　时间元组各项含义

索　引	属　性	含　义
0	tm_year	年
1	tm_mon	月
2	tm_mday	日
3	tm_hour	时
4	tm_min	分
5	tm_sec	秒
6	tm_wday	一周中的第几天
7	tm_yday	一年中的第几天
8	tm_isdst	夏令时

```
>>> t = time.localtime()
>>> t[1]
5
```

通过索引能够得到相应的属性，上面的例子中就得到了当前时间的月份。

其实，time.localtime()不是没有参数，它在默认情况下，以time.time()的时间戳为参数。言外之意就是说可以自己输入一个时间戳，返回那个时间戳所对应的时间（按照公元和时分秒计时）。例如：

```
>>> time.localtime(100000)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=2, tm_hour=11, tm
```

- gmtime()

localtime()得到的是本地时间，如果要国际化，就最好使用格林威治时间。可以这样：

```
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=4, tm_hour=23, tm
```

格林威治标准时间是指位于英国伦敦郊区的皇家格林威治天文台的标准时间，因为本初子午线被定义在通过那里的经线。

还有更友好的，请继续阅读。

- asctime()

```
>>> time.asctime()
'Mon May  4 21:46:13 2015'
```

time.asctime()的参数为空时，默认是以time.localtime()的值为参数，所以得到的是当前日期时间和星期。当然，也可以自己设置参数：

```
>>> h = time.localtime(1000000)
>>> h
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=12, tm_hour=21, t
>>> time.asctime(h)
'Mon Jan 12 21:46:40 1970'
```

注意，time.asctime()的参数必须是时间元组，类似上面那种。若不是时间戳，通过time.time()得到的时间戳也可以转化为上面的形式。

- ctime()

```
>>> time.ctime()
'Mon May  4 21:52:22 2015'
```

在没有参数的时候，事实上是以time.time()的时间戳为参数，也可以自定义一个时间戳。

```
>>> time.ctime(1000000)
'Mon Jan 12 21:46:40 1970'
```

跟前面得到的结果是一样的，只不过用了时间戳作为参数。

在前述函数中，通过localtime()、gmtime()得到的是时间元组，通过time()得到的是时间戳。有的函数如asctime()是以时间元组为参数，有的如ctime()是以时间戳为函数，这样做的目的是为了满足编程中多样化的需要。

- mktime()

mktime()也是以时间元组为参数，但是它返回的不是可读性更好的那种样式，而是：

```
>>> lt = time.localtime()
>>> lt
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=5, tm_hour=7, tm_
>>> time.mktime(lt)
1430783729.0
```

返回了时间戳，类似于localtime()的逆过程（localtime()是以时间戳为参数）。

好像还缺点什么，因为在编程中，使用比较多的是“字符串”，似乎还没有将时间转化为字符串的函数，这个应该有。

- strftime()

函数格式稍微复杂一些。

```
Help on built-in function strftime in module time:

strftime(...) strftime(format[, tuple]) -> string

Convert a time tuple to a string according to a format specifi
```

将时间元组按照指定格式要求转化为字符串。如果不指定时间元组，就默认为localtime()值。说其复杂是在于其format，需要用到下面的东西，如表6-2所示。

表6-2　format格式定义

格式	含　义	取值范围（格式）
%y	去掉世纪的年份	00～99，如“15”
%Y	完整的年份	如“2015”
%j	指定日期是一年中的第几天	001～366
%m	返回月份	01～12
%b	本地简化月份的名称	简写英文月份
%B	本地完整月份的名称	完整英文月份
%d	该月的第几日	如5月1日返回“01”
%H	该日的第几时（24小时制）	00～23
%l	该日的第几时（12小时制）	01～12
%M	分钟	00～59
%S	秒	00～59
%U	在该年中的第多少星期（以周日为一周起点）	00～53
%W	同上，只不过是以周一为起点	00～53
%w	一星期中的第几天	0～6
%Z	时区	中国返回CST，即China Standard Time
%x	日期	日/月/年
%X	时间	时:分:秒
%c	详细日期时间	日/月/年 时:分:秒
%%	‘%’字符	‘%’字符
%p	上下午	AM or PM

简要列举如下：

```
>>> time.strftime("%y,%m,%d")
'15,05,05'
>>> time.strftime("%y/%m/%d")
'15/05/05'
```

分隔符可以自由指定，既然已经变成字符串了，就可以“随心所欲不逾矩”了。

- strptime()

```
    Help on built-in function strptime in module time:

    strptime(...) strptime(string, format) -> struct_time

    Parse a string to a time tuple according to a format specifica
```

strptime()的作用是将字符串转化为时间元组。请注意，其参数要指定两个，一个是时间字符串，另外一个是时间字符串所对应的格式，格式符号用表6-2中的。例如：

```
>>> today = time.strftime("%y/%m/%d")
>>> today
'15/05/05'
>>> time.strptime(today, "%y/%m/%d")
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=5, tm_hour=0, tm_
```

6.3.8 datetime

虽然time模块已经能够把有关时间方面的东西搞定了，但是，有时调用起来感觉不是很直接，于是又出来了一个datetime模块，供程序员们选择使用。

datetime模块中有以下几个类。

- datetime.date：日期类，常用的属性有year/month/day。
- datetime.time：时间类，常用的有hour/minute/second/microsecond。
- datetime.datetime：日期时间类。
- datetime.timedelta：时间间隔，即两个时间点之间的时间长度。
- datetime.tzinfo：时区类。

1.date类

通过实例了解常用的属性：

```
>>> import datetime
>>> today = datetime.date.today()
>>> today
datetime.date(2015, 5, 5)
```

其实这里生成了一个日期对象，然后操作这个对象的各种属性。用print语句，可以使视觉更佳：

```
>>> print today
2015-05-05
>>> print today.ctime()
Tue May  5 00:00:00 2015
>>> print today.timetuple()
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=5, tm_hour=0, tm_
>>> print today.toordinal()
735723
```

特别注意，如果你妄图用datetime.date.year()是会报错的，因为year不是一个方法，必须这样：

```
>>> print today.year
2015
>>> print today.month
5
>>> print today.day
5
```

进一步看看时间戳与格式化时间格式的转换：

```
>>> to = today.toordinal()
>>> to
735723
>>> print datetime.date.fromordinal(to)
2015-05-05

>>> import time
>>> t = time.time()
>>> t
1430787994.80093
>>> print datetime.date.fromtimestamp(t)
2015-05-05
```

还可以更灵活一些，修改日期。

```
>>> d1 = datetime.date(2015,5,1)
>>> print d1
2015-05-01
>>> d2 = d1.replace(year=2005, day=5)
>>> print d2
2005-05-05
```

2.time类

也要生成time对象。

```
>>> t = datetime.time(1,2,3)
>>> print t
01:02:03
```

它的常用属性：

```
>>> print t.hour
1
>>> print t.minute
2
>>> print t.second
3
>>> t.microsecond
0
>>> print t.tzinfo
None
```

3.timedelta类

主要用来做时间的运算。比如：

```
>>> now = datetime.datetime.now()
>>> print now
2015-05-05 09:22:43.142520
```

没有讲述datetime类，因为在有了date和time类知识之后，这个类比较简单。我最喜欢now方法，对now增加5个小时：

```
>>> b = now + datetime.timedelta(hours=5)
>>> print b
2015-05-05 14:22:43.142520
```

增加两周：

```
>>> c = now + datetime.timedelta(weeks=2)
>>> print c
2015-05-19 09:22:43.142520
```

计算时间差：

```
>>> d = c - b
>>> print d
13 days, 19:00:00
```

6.3.9 urllib

urllib模块用于读取来自网上（服务器上）的数据，比如很多人用Python做爬虫程序，就可以使用这个模块。先看一个简单例子：

```
>>> import urllib
>>> itdiffer =  urllib.urlopen("http://www.itdiffer.com")
```

这样就已经把我的网站www.itdiffer.com首页的内容拿过来了，得到了一个类似文件的对象。接下来的操作跟操作一个文件一样。

```
>>> print itdiffer.read()
<!DOCTYPE HTML>
<html>
    <head>
        <title>I am Qiwsir</title>
....//因为内容太多，下面就省略了
```

这样就完成了对网页的一个抓取。当然，如果你真的要做爬虫程序，还不是仅仅如此。这里不介绍爬虫程序如何编写（关于爬虫的资料，网上已经有很多了，读者可以搜索并学习，如果实在理解有困难，还可以跟我联系，我会协助你的），仅说明urllib模块的常用属性和方法。

```
>>> dir(urllib)
['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLoper
```

选几个常用的介绍，如果读者用到其他的，可以通过查看文档了

解。

1.urlopen（url，data=None，proxies=None）

urlopen()主要用于打开url文件，从而获得指定url网页内容，然后就如同操作文件那样来操作。

```
Help on function urlopen in module urllib:

urlopen(url, data=None, proxies=None) Create a file-like objec
```

得到的对象被叫作“类文件”，从名字中也可以理解后面的操作了。先对参数说明一下。

- url：远程数据的路径，常常是网址。
- data：如果使用post方式，这里就是所提交的数据。
- proxies：设置代理。

关于参数的详细说明，还可以参考官方文档，这里仅演示最常用的，如前面的例子那样。

当得到了类文件对象之后，即变量itdiffer引用了得到的类文件对象，这个对象依然可以用老办法来查看它的属性和方法。

```
>>> dir(itdiffer)
['__doc__', '__init__', '__iter__', '__module__', '__repr__', 'clo
```

从结果中也可以看出，这个类文件对象也是可迭代的，下面是比较常用的方法。

- read()、readline()、readlines()、fileno()、close()：都与文件操作一样，这里不再赘述。
- info()：返回头信息。
- getcode()：返回http状态码。
- geturl()：返回url。

简单举例：

```
>>> itdiffer.info()
<httplib.HTTPMessage instance at 0xb6eb3f6c>
>>> itdiffer.getcode()
```

```
200
>>> itdiffer.geturl()
'http://www.itdiffer.com'
```

许多时候，在建立了类文件对象后，要通过其方法得到某些数据。

2.对url编码、解码

url对其中的字符有严格要求，不许可某些特殊字符直接使用某些字符，比如url中有空格，会自动将空格进行处理，这个过程需要对url进行编码和解码。在进行web开发的时候要特别注意这里。

urllib模块提供了url编码和解码功能。

- quote（string[，safe]）：对字符串进行编码。参数safe指定了不需要编码的字符。
- urllib.unquote（string）：对字符串进行解码。
- quote_plus（string[，safe]）：与urllib.quote类似，但这个方法用“+”来替换空格，而quote用“%20”来代替空格。
- unquote_plus（string）：对字符串进行解码。
- urllib.urlencode（query[，doseq]）：将dict或者包含两个元素的元组列表转换成url参数。例如{'name':'laoqi'，'age':40}将被转换为“name=laoqi&age=40”。
- pathname2url（path）：将本地路径转换成url路径。
- url2pathname（path）：将url路径转换成本地路径。

看例子就更明白了：

```
>>> du = "http://www.itdiffer.com/name=python book"
>>> urllib.quote(du)
'http%3A//www.itdiffer.com/name%3Dpython%20book'
>>> urllib.quote_plus(du)
'http%3A%2F%2Fwww.itdiffer.com%2Fname%3Dpython+book'
```

注意看空格的变化，一个被编码成“%20”，另外一个是“+”。

再看解码的，假如在Google中搜索《零基础学Python》，结果如图6-5所示。



图6-5　在Google中搜索《零基础学Python》

与本书同步的网络教程在这次搜索中排列第一个哦。

这不是重点，重点是看url，它就是用“＋”替代空格。

```
>>> dup = urllib.quote_plus(du)
>>> urllib.unquote_plus(dup)
'http://www.itdiffer.com/name=python book'
```

从解码效果来看，是比较完美的逆过程。

```
>>> urllib.urlencode({"name":"qiwsir","web":"itdiffer.com"})
'web=itdiffer.com&name=qiwsir'
```

如果将来你要做一个网站，上面的这个方法或许会用到的。

- urlretrieve()

虽然urlopen()能够建立类文件对象，但是，不等于将远程文件保存在本地存储器中，urlretrieve()就是满足这个需要的。先看实例：

```
>>> import urllib
>>> urllib.urlretrieve("http://www.itdiffer.com/images/me.jpg","me
('me.jpg', <httplib.HTTPMessage instance at 0xb6ecb6cc>)
>>>
```

me.jpg是一张存在于服务器上的图片，地址是：http://www.itdiffer.com/images/me.jpg，把它保存到本地存储器中，并且

仍命名为me.jpg。注意，如果只写这个名字，表示存在启动Python交互模式的那个目录中，否则，可以指定存储具体目录和文件名。

在urllib官方文档中有一大段相关说明，读者可以去认真阅读。这里仅简要介绍一下相关参数。

```
urllib.urlretrieve(url[, filename[, reporthook[, data]]])
```

- url：文件所在的网址。
- filename：可选。将文件保存到本地的文件名，如果不指定，urllib会生成一个临时文件来保存。
- reporthook：可选。是回调函数，当链接服务器和相应数据传输完毕时触发本函数。
- data：可选。用post方式所发出的数据。

函数执行完毕，返回的结果是一个元组（filename，headers），filename是保存到本地的文件名，headers是服务器响应头信息。

```
#!/usr/bin/env python
# coding=utf-8

import urllib

def go(a, b, c):
    per = 100.0 * a * b / c
    if per > 100:
        per = 100
    print "%.2f%%" % per

url = "http://youxi.66wz.com/uploads/1046/1321/11410192.90d133701b
local = "/home/qw/Pictures/g.jpg"
urllib.urlretrieve(url, local, go)
```

这段程序就是要下载指定的图片，并且保存为本地指定位置的文件，同时要显示下载的进度。上述文件保存之后执行，显示如下效果：

```
$ python 22501.py
0.00%
8.13%
16.26%
24.40%
32.53%
40.66%
```

```
48.79%
56.93%
65.06%
73.19%
81.32%
89.46%
97.59%
100.00%
```

到相应目录中查看，能看到与网上地址一样的文件。这里就不对结果截图了，读者自行查看（或许在本书出版的时候，这张神秘的图片你已经看不到了，你应该把这视为正常的事情，那么你就换一张图片地址吧）。

6.3.10 urllib2

urllib2是另外一个模块，它跟urllib有相似的地方——都是对url相关的操作，也有不同的地方。

有时候两个要同时使用，urllib模块和urllib2模块有的方法可以相互替代，有的不能。看下面的属性方法列表就知道了。

```
>>> dir(urllib2)
['AbstractBasicAuthHandler', 'AbstractDigestAuthHandler', 'Abstrac
```

读者不妨将urllib和urllib2的方法属性名称进行比较，会发现它们之间有一部分是相同的，比如urlopen()跟urllib.open()非常类似，除了相同的就是不同的了。

Request类

在前面引用的内容中就明确指出，利用urllib2模块可以建立一个Request对象，建立Request对象的方法就是使用Request类。

```
>>> req = urllib2.Request("http://www.itdiffer.com")
```

建立了Request对象之后，它的最直接应用可以作为urlopen()方法的参数。

```
>>> response = urllib2.urlopen(req)
>>> page = response.read()
>>> print page
```

因为与前面的urllib.open（"http://www.itdiffer.com"）结果一样，就不再赘述。

但是，如果Request对象仅仅局限于此，似乎还没有什么太大的优势。因为刚才的访问仅仅满足以get方式请求页面，并建立类文件对象。如果是通过post向某地址提交数据，也可以建立Request对象。

```
import urllib
import urllib2

url = 'http://www.itdiffer.com/register.py'

values = {'name' : 'qiwsir',
          'location' : 'China',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)         #发送请求同时传data表单
response = urllib2.urlopen(req)          #接受反馈的信息
the_page = response.read()               #读取反馈的内容
```

如果读者照抄上面的程序，然后运行代码，肯定是报错的，因为那个url中没有相应的接受客户端post上去的data的程序文件，为了让程序运行，读者可以开发接收数据的程序。上面的代码只是以一个例子来显示Request对象的另外一个用途，并且在这个例子中以post方式提交数据。

在网站中，有的会通过User-Agent来判断访问者是浏览器还是别的程序，如果通过别的程序访问，它有可能拒绝。这时候我们编写程序去访问，就要设置headers了。设置方法是：

```
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
headers = { 'User-Agent' : user_agent }
```

然后重新建立Request对象：

```
req = urllib2.Request(url, data, headers)
```

再用urlopen()方法访问：

```
response = urllib2.urlopen(req)
```

除了上面的演示之外，urllib2模块的东西还有很多，比如还可以：

- 设置HTTP Proxy
- 设置Timeout值
- 自动redirect
- 处理cookie

这些内容不再一一介绍，当需要用到的时候可以查看文档或者去网上搜索。

6.3.11　XML

XML在软件领域用途非常广泛，有名人曰：

“当XML（扩展标记语言）于1998年2月被引入软件工业界时，它给整个行业带来了一场风暴。有史以来第一次，这个世界拥有了一种用来结构化文档和数据的通用且适应性强的格式，它不仅仅可以用于Web，而且可以被用于任何地方。”

——《Designing With Web Standards Second Edition》，Jeffrey Zeldman

如果要对XML做一个定义式的说明，就不得不引用w3school里面简洁而明快的说明：

- XML指可扩展标记语言（EXtensible Markup Language）。
- XML是一种标记语言，很类似于HTML。
- XML的设计宗旨是传输数据，而非显示数据。
- XML标签没有被预定义，你需要自行定义标签。
- XML被设计为具有自我描述性。
- XML是W3C的推荐标准。

如果读者要详细了解和学习XML，可以阅读w3school的教程。

XML的重要在于它是用来传输数据的，因此，特别是在Web编程中，经常要用到。有了它让数据传输变得简单了，这么重要，Python

当然支持。

一般来讲，一个引人关注的东西，总会有很多人从不同侧面去关注。在编程语言中也是如此，所以，对XML这个明星似的东西，Python提供了多种模块来处理。

- xml.dom.*模块：Document Object Model。适合用于处理DOM API。它能够将XML数据在内存中解析成一棵树，然后通过对树的操作来操作XML，但是，由于这种方式将XML数据映射到内存中的树，导致比较慢，且消耗更多内存。
- xml.sax.*模块：simple API for XML。由于SAX以流式读取XML文件，从而速度较快，且少占用内存，但是操作上稍复杂，需要用户实现回调函数。
- xml.parser.expat：是一个直接的，低级一点的基于C的expat的语法分析器。expat接口基于事件反馈，有点像SAX但又不太像，因为它的接口并不是完全规范于expat库的。
- xml.etree.ElementTree（以下简称ET）：元素树。它提供了轻量级的Python式的API，相对于DOM，ET快了很多，而且有很多令人愉悦的API可以使用；相对于SAX，ET也有ET.iterparse提供了“在空中”的处理方式，没有必要加载整个文档到内存，节省内存。ET性能的平均值和SAX差不多，但是API的效率更高一点而且使用起来很方便。

所以，我用xml.etree.ElementTree。

ElementTree在标准库中有两种实现，一种是纯Python实现：xml.etree.ElementTree，另外一种是：xml.etree.cElementTree。

如果读者使用的是Python 2.x，可以像这样引入模块：

```
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET
```

如果是Python 3.3以上，就没有这个必要了，只需要一句话import xml.etree.ElementTree as ET即可，然后由模块自动来寻找适合的方式。显然Python 3.x相对Python 2.x有了很大进步。

1.遍历查询

先要做一个XML文档，用w3school中的一个例子，如图6-6所示。

这是一个XML树，只不过是用图来表示的，先把这棵树写成XML文档格式。

```
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
    </book>
        <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
</bookstore>
```

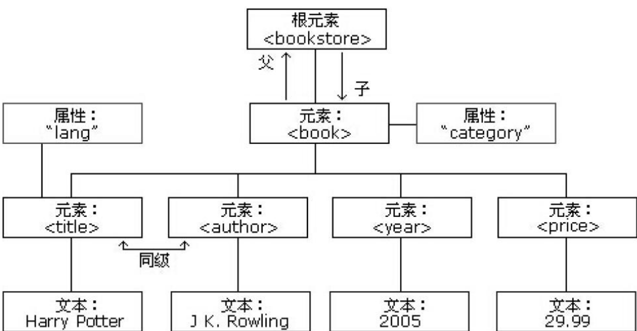


图6-6　XML树

将XML保存为名为22601.xml的文件，接下来就是以它为对象，练习各种招数了。

```
>>> import xml.etree.cElementTree as ET
```

为了简化，用这种方式引入，在编程实践中推荐读者使用try...except...方式。

```
>>> tree = ET.ElementTree(file="22601.xml")
>>> tree
<ElementTree object at 0xb724cc2c>
```

建立起XML解析树，然后通过根节点向下开始读取各个元素（element对象）。

在上述XML文档中，根元素是bookstore，它没有属性，或者属性为空。

```
>>> root = tree.getroot()       #获得根
>>> root.tag
'bookstore'
>>> root.attrib
{}
```

要想将根下面的元素都读出来，可以：

```
>>> for child in root:
...     print child.tag, child.attrib
...
book {'category': 'COOKING'}
book {'category': 'CHILDREN'}
book {'category': 'WEB'}
```

也可以用下面方法读取指定元素的信息：

```
>>> root[0].tag
'book'
>>> root[0].attrib
{'category': 'COOKING'}
>>> root[0].text              #无内容
'\n        '
```

再深入一层，就有内容了：

```
>>> root[0][0].tag
'title'
>>> root[0][0].attrib
{'lang': 'en'}
>>> root[0][0].text
```

```
'Everyday Italian'
```

对于ElementTree对象，有一个iter方法可以对指定名称的子节点进行深度优先遍历。例如：

```
>>> for ele in tree.iter(tag="book"):          #遍历名称为book的节点
...     print ele.tag, ele.attrib
...
book {'category': 'COOKING'}
book {'category': 'CHILDREN'}
book {'category': 'WEB'}

>>> for ele in tree.iter(tag="title"):          #遍历名称为title的节
...     print ele.tag, ele.attrib, ele.text
...
title {'lang': 'en'} Everyday Italian
title {'lang': 'en'} Harry Potter
title {'lang': 'en'} Learning XML
```

如果不指定元素名称，就是将所有的元素遍历一遍。

```
>>> for ele in tree.iter():
...     print ele.tag, ele.attrib
...
bookstore {}
book {'category': 'COOKING'}
title {'lang': 'en'}
author {}
year {}
price {}
book {'category': 'CHILDREN'}
title {'lang': 'en'}
author {}
year {}
price {}
book {'category': 'WEB'}
title {'lang': 'en'}
author {}
year {}
price {}
```

除了上面的方法，还可以通过路径搜索到指定的元素，读取其内容，这就是xpath。此处对xpath不详解，如果要了解可以到网上搜索有关信息。

```
>>> for ele in tree.iterfind("book/title"):
```

```
...     print ele.text
...
Everyday Italian
Harry Potter
Learning XML
```

利用findall()方法，也可以实现查找功能：

```
>>> for ele in tree.findall("book"):
...     title = ele.find('title').text
...     price = ele.find('price').text
...     lang = ele.find('title').attrib
...     print title, price, lang
...
Everyday Italian 30.00 {'lang': 'en'}
Harry Potter 29.99 {'lang': 'en'}
Learning XML 39.95 {'lang': 'en'}
```

2.编辑

除了读取有关数据之外，还能对XML进行编辑，即增、删、改、查功能，还是以上面的XML文档为例：

```
>>> root[1].tag
'book'
>>> del root[1]
>>> for ele in root:
...     print ele.tag
```

如此，成功删除了一个节点，原来有三个book节点，现在就还剩两个了。打开源文件再看看，是不是正好少了第二个节点呢？一定很让你失望，源文件居然没有变化。

的确如此，源文件没有变化，因为至此的修改动作，还停留在内存中，还没有将修改结果输出到文件。不要忘记，我们是在内存中建立的ElementTree对象。再这样做：

```
>>> import os
>>> outpath = os.getcwd()
>>> file = outpath + "/22601.xml"
```

把当前文件路径拼装好。然后：

```
```

```
>>> tree.write(file)
```

再看源文件，已经变成两个节点了。

除了删除，也能够修改：

```
>>> for price in root.iter("price"):          #原来每本书的价格
...     print price.text
...
30.00
39.95
>>> for price in root.iter("price"):          #每本上涨7元，并且增加属性
...     new_price = float(price.text) + 7
...     price.text = str(new_price)
...     price.set("updated","up")
...
>>> tree.write(file)
```

查看源文件：

```
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price updated="up">37.0</price>
    </book>
    <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price updated="up">46.95</price>
    </book>
</bookstore>
```

不仅价格修改了，而且在price标签里面增加了属性标记。

上面用del来删除某个元素，其实，在编程中用的不多，较多使用remove()方法。比如要删除price＞40的书。可以这么做：

```
>>> for book in root.findall("book"):
...     price = book.find("price").text
...     if float(price) > 40.0:
...         root.remove(book)
...
>>> tree.write(file)
```

于是就这样了：

```
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price updated="up">37.0</price>
    </book>
</bookstore>
```

接下来就要增加元素了。

```
>>> import xml.etree.cElementTree as ET
>>> tree = ET.ElementTree(file="22601.xml")
>>> root = tree.getroot()
>>> ET.SubElement(root, "book")          #在root里面添加book节点
<Element 'book' at 0xb71c7578>
>>> for ele in root:
...     print ele.tag
...
book
book
>>> b2 = root[1]                          #得到新增的book节点
>>> b2.text = "python"                    #添加内容
>>> tree.write("22601.xml")
```

查看源文件：

```
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price updated="up">37.0</price>
    </book>
    <book>python</book>
</bookstore>
```

3.常用属性和方法总结

ET里面的属性和方法不少，这里列出常用的，供使用中备查。

（1）Element对象

常用属性如下。

- tag：string，元素数据种类。
- text：string，元素的内容。
- attrib：dictionary，元素的属性字典。
- tail：string，元素的尾形。

针对属性的操作如下。

- clear()：清空元素的后代、属性、text和tail也设置为None。
- get（key，default=None）：获取key对应的属性值，如该属性不存在则返回default值。
- items()：根据属性字典返回一个列表，列表元素为（key，value）。
- keys()：返回包含所有元素属性键的列表。
- set（key，value）：设置新的属性键与值。

针对后代的操作如下。

- append（subelement）：添加直系子元素。
- extend（subelements）：增加一串元素对象作为子元素。
- find（match）：寻找第一个匹配子元素，匹配对象可以为tag或path。
- findall（match）：寻找所有匹配子元素，匹配对象可以为tag或path。
- findtext（match）：寻找第一个匹配子元素，返回其text值。匹配对象可以为tag或path。
- insert（index，element）：在指定位置插入子元素。
- iter（tag=None）：生成遍历当前元素所有后代或者给定tag的后代的迭代器。
- iterfind（match）：根据tag或path查找所有的后代。
- itertext()：遍历所有后代并返回text值。
- remove（subelement）：删除子元素。

（2）ElementTree对象

- find（match）。
- findall（match）。
- findtext（match，default=None）。
- getroot()：获取根节点。
- iter（tag=None）。

- iterfind（match）。
- parse（source，parser＝None）：装载xml对象，source可以为文件名或文件类型对象。
- write（file，encoding＝"us-ascii"，xml_declaration＝None，default_namespace＝None，method＝"xml"）。

6.3.12 JSON

就传递数据而言，XML是一种选择，还有另外一种——JSON，它是一种轻量级的数据交换格式，如果读者要做Web编程，则会用到它。根据维基百科的相关内容，对JSON了解一下：

JSON（JavaScript Object Notation）是一种由道格拉斯·克罗克福特构想设计、轻量级的数据交换语言，以文字为基础，且易于让人阅读。尽管JSON是JavaScript的一个子集，但JSON是独立于语言的文本格式，并且采用了类似于C语言家族的一些习惯。

关于JSON更为详细的内容，可以参考网站：http://www.json.org。

从上述网站摘取部分内容，了解一下JSON的结构。

JSON建构于两种结构：

- "名称/值"对的集合（A collection of name/value pairs），不同的语言中，它被理解为对象（object）、纪录（record）、结构（struct）、字典（dictionary）、哈希表（hash table）、有键列表（keyed list）或者关联数组（associative array）。
- 值的有序列表（An ordered list of values），在大部分语言中，它被理解为数组（array）。

Python标准库中有JSON模块，主要执行序列化和反序列化功能。

- 序列化：encoding，把一个Python对象编码转化成JSON字符串。
- 反序列化：decoding，把JSON格式字符串解码转换为Python数据对象。

1.基本操作

JSON模块相对XML单纯了很多：

```
>>> import json
>>> json.__all__
['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONEncoder']
```

- encoding: dumps()

```
>>> data = [{"name":"qiwsir", "lang":("python", "english"), "age":
>>> print data
[{'lang': ('python', 'english'), 'age': 40, 'name': 'qiwsir'}]
>>> data_json = json.dumps(data)
>>> print data_json
[{"lang": ["python", "english"], "age": 40, "name": "qiwsir"}]
```

encoding的操作是比较简单的，请注意观察data和data_json的不同——lang的值从元组变成了列表，还有不同：

```
>>> type(data_json)
<type 'str'>
>>> type(data)
<type 'list'>
```

- decoding: loads()

decoding的过程也像上面一样简单：

```
>>> new_data = json.loads(data_json)
>>> new_data
[{u'lang': [u'python', u'english'], u'age': 40, u'name': u'qiwsir'
```

需要注意的是，解码之后并没有将元组还原。

上面的data都不是很长，还能凑合阅读，如果很长了，阅读就有难度了。所以，JSON的dumps()提供了可选参数，利用它们能在输出上对人更友好（这对机器是无所谓的）。

```
>>> data_j = json.dumps(data, sort_keys=True, indent=2)
>>> print data_j
[
  {
```

```
    "age": 40,
    "lang": [
      "python",
      "english"
    ],
    "name": "qiwsir"
  }
]
```

sort_keys=True意思是按照键的字典顺序排序，indent=2是让每个键/值对显示的时候，以缩进两个字符对齐，这样的视觉效果好多了。

2.大JSON字符串

如果数据不是很大，那么上面的操作足够了，但现在是“大数据”时代了，随便一个什么业务都在说自己是大数据，显然不能总让JSON很小。前面的操作方法是将数据都读入内存，如果数据量太大了内存会爆满，这肯定是不行的。怎么办？JSON提供了load()函数和dump()函数解决这个问题，注意，跟已经用过的函数相比是不同的，请仔细观察。

```
>>> import tempfile                    #临时文件模块
>>> data
[{'lang': ('python', 'english'), 'age': 40, 'name': 'qiwsir'}]
>>> f = tempfile.NamedTemporaryFile(mode='w+')
>>> json.dump(data, f)
>>> f.flush()
>>> print open(f.name, "r").read()
[{"lang": ["python", "english"], "age": 40, "name": "qiwsir"}]
```

6.4 第三方库

标准库的内容已经非常多了，前面仅仅列举几个，但是Python给编程者的支持不仅仅在于标准库，还有不可胜数的第三方库。因此，作为一个Pythoner，即使你达到了master的水平，在做某个事情之前最好在网上搜索一下是否有标准库或者第三方库能替你完成。因为，伟大的艾萨克·牛顿爵士说过：如果我比别人看得更远，那是因为我站在巨人的肩上。

编程就更要站在巨人的肩上，标准库和第三方库以及其提供者就是巨人，我们本应当谦卑地向其学习，并应用其成果。

6.4.1 安装第三方库

安装第三方库的方法有几种，不同方法有不同的优缺点，读者可以根据自己的喜好或者实际的工作情景来选择使用。

方法一：利用源码安装

在github.com网站可以下载第三方库的源码（注意，github不是源码的唯一来源，只不过很多源码都在这个网站上，我也喜欢罢了），得到源码之后，在本地安装。

如果你下载的是一个文件包，即得到的源码格式为zip或tar.zip或tar.bz2的压缩文件，需要先解压缩，然后进入其目录（文件夹）；如果你能熟练使用git命令，可以直接从github中clone源码到本地计算机上，然后再进入该目录（文件夹）。进去之后通常会看见setup.py文件。如果是Linux操作系统或者苹果计算机（我是用Ubuntu系统，特别推荐哦），就在这里运行shell，执行命令：

```
python setup.py install
```

如果用的是Windows系统，需要打开命令行模式，执行上述指令即可。

如此，就能把这个第三库安装到系统里。具体位置，要视操作系统和你当初安装Python环境时设置的路径而定。默认条件下，Windows是在C:\Python2.7\Lib\site-packages，Linux在/usr/local/lib/python2.7/dist-packages（这个只是参考，不同发行版会有差别，具体请读者根据自己的操作系统找找），Mac在/Library/Python/2.7/site-packages。

这种安装的方法有时候麻烦一些，但是比较灵活，主要体现在：

- 可以下载安装自己选定的任意版本的第三方库，比如最新版，或者更早的某个版本，所以在某些有特殊需要的时候，常常使用这种方式安装第三方库。
- 通过安装设置可以指定安装目录，自由度比较高。

有安装就要有卸载，卸载所安装的库非常简单，只需要到相应系统的site-packages目录，直接删掉库文件即可。

方法二：pip安装

用源码安装，不是我推荐的，我推荐的是用第三方库的管理工具安装。

有一个网站专门用来存储第三方库，在这个网站上的所有软件包，都能用pip或者easy_install这种安装工具来安装，网站的地址：https://pypi.python.org/pypi。

pip是一个以Python计算机程序语言写成的软件包管理系统，可以安装和管理软件包，另外很多软件包也可以在“Python软件包索引”（英语：Python Package Index，简称PyPI）中找到。（源自《维基百科》）

首先，要安装pip。如果读者跟我一样，用的是Ubuntu系统或者其他某种Linux系统，就用不到这个操作，因为在安装操作系统的时候已经默认把这个东西安装好了。如果因为什么原因而没有安装，可以使用如下方法：

- Debian and Ubuntu：

```
sudo apt-get install python-pip
```

- Fedora and CentOS：

```
sudo yum install python-pip
```

当然，也可以下载文件get-pip.py，然后执行python get-pip.py来安装，下载地址是：https://bootstrap.pypa.io/get-pip.py。这个方法也适用于Windows系统。

对于Windows操作系统，如果你安装了某个版本的Python，特别注意到安装目录中找一找，一般情况下pip就已经默认安装好了。

pip就这样安装好了，非常简单吧。

然后你就可以淋漓尽致地安装第三方库了，之所以那么痛快，是因为只需要执行pip install XXXXXX（XXXXXX代表第三方库的名字）即可。

第三方库安装完毕。

6.4.2 以requests为例

以requests模块为例来说明第三方库的安装和使用。之所以选这个，是因为前面介绍了urllib和urllib2两个标准库的模块，与之有类似功能的第三方库中requests也是一个用于在程序中进行http协议下的get和post请求的模块，并且被网友说成“好用的要哭”。

说明：下面的内容是根据网友1world0x00提供的文章修改而成的，对她表示万分感激。

1.安装

```
pip install requests
```

安装好之后，在交互模式下：

```
>>> import requests
>>> dir(requests)
['ConnectionError', 'HTTPError', 'NullHandler', 'PreparedRequest',
```

从上面的列表中可以看出，在http中常用到的get、cookies、post等都赫然在目。

2.get请求

```
>>> r = requests.get("http://www.itdiffer.com")
```

得到一个请求的实例，然后：

```
>>> r.cookies
<<class 'requests.cookies.RequestsCookieJar'>[]>
```

这个网站对客户端没有写任何cookies内容，换一个看看：

```
>>> r = requests.get("http://www.1world0x00.com")
>>> r.cookies
<<class 'requests.cookies.RequestsCookieJar'>[Cookie(version=0, na
```

仔细观察，是不是看到了cookie的name和value，结合对网络有关知识的了解，是不是有一种豁然开朗恍然大悟的感觉？

继续，还有别的属性可以看看：

```
>>> r.headers
{'x-powered-by': 'PHP/5.3.3', 'transfer-encoding': 'chunked', 'set

>>> r.encoding
'UTF-8'

>>> r.status_code
200
```

这些都是在客户端看到的网页的基本属性。

下面这个比较长，是网页的内容，仅仅截取显示的部分：

```
>>> print r.text

<!DOCTYPE html>
<html lang="zh-CN">
  <head>
  <meta charset="utf-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale
  <title>1world0x00sec</title>
  <link rel="stylesheet"
href="http://www.1world0x00.com/usr/themes/default/style.min.css">
  <link rel="canonical" href="http://www.1world0x00.com/" />
  <link rel="stylesheet" type="text/css"
href="http://www.1world0x00.com/usr/plugins/CodeBox/css/codebox.cs
  <meta name="description" content="爱生活，爱拉芳。不装逼还能做朋友。"
  <meta name="keywords" content="php" />
  <link rel="pingback" href="http://www.1world0x00.com/index.php/a

  ......
```

请求发出后，requests会基于http头部对相应的编码做出有根据的推测，当你访问r.text时，requests使用其推测的文本编码。你可以找出requests使用了什么编码，并且能够使用r.coding属性来改变它。

```
>>> r.content
'\xef\xbb\xbf\xef\xbb\xbf<!DOCTYPE html>\n<html lang="zh-CN">\n  <
```

以二进制的方式打开服务器并返回数据。

3.post请求

假如你要向某个服务器发送一些数据，一般情况下，使用的就是post，实现方式也比较简单，只需要传递一个字典给data参数。

```
>>> import requests
>>> payload = {"key1":"value1", "key2":"value2"}
>>> r = requests.post("http://httpbin.org/post")
>>> r1 = requests.post("http://httpbin.org/post", data=payload)
```

r没有提供data参数值，得到的效果是：

```
{
    "args": {},
    "data": {},
    "form": {},
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Connection": "close",
        "Content-Length": "0",
        "Host": "httpbin.org",
        "User-Agent": "python-requests/2.4.3 Cpython/2.7.8 Windows/7",
        "X-Request-Id": "19ed80fc-ffe6-4dc0-b83a-08dba09daf88"
    },
    "json": null,
    "origin": "118.113.116.160",
    "url": "http://httpbin.org/post"
}
```

r1为data提供了值，再看效果：

```
{
    "args": {},
    "data": {},
    "form": {
        "key1": "value1",
        "key2": "value2"
    },
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Connection": "close",
        "Content-Length": "23",
        "Connent-Type": "application/x-www-form-urlencoded",
        "Host": "httpbin.org",
        "User-Agent": "python-requests/2.4.3 Cpython/2.7.8 Windows/7",
        "X-Request-Id": "b8ba897f-44c9-4922-b157-562e0cf07bcd"
    },
    "json": null,
    "origin": "118.113.116.160",
    "url": "http://httpbin.org/post"
}
```

新闻比较看才有意思，代码也如此。比较上面的两个截图，发现后者当data被赋值之后，在结果中多的form值，就多了data所传入的值，form的值就是post给服务器的内容。

4.http头部

```
>>> r.headers['content-type']
'application/json'
```

注意，引号里面的内容不区分大小写“CONTENT-TYPE”也可以，也能够自定义头部。

```
>>> r.headers['content-type'] = 'adad'
>>> r.headers['content-type']
'adad'
```

注意，当定制头部的时候，如果需要定制的项目有很多，一般用到字典类型的数据。

通过一个实例，展示第三方模块的应用方法，其实没有什么特殊的地方，只要安装了，就和用标准库模块一样了。

根据我的个人经验，第三方模块常常在某个方面做得更友好，或者性能更优化，所以，不要将其放在我们的视野之外。

第7章　保存数据

如果在程序中，有数据要保存到磁盘中，放到某个文件中是一种不错的方法。但是，要注意存到文件中的数据不能随意放置，因为你早晚还要把数据读出来，如果没有什么规律，读出来的时候会很麻烦。

很多开发者早就意识到这点了，于是就出现了将要存储的对象格式化（或者叫作序列化），即按照某种特定要求将对象存到文件中，才好存好取，这有点类似集装箱的作用。

7.1 pickle

pickle是标准库中的一个模块，还有跟它完全一样的叫作cpickle，两者的区别就是后者更快（似乎已经是一个规律了，凡是某个模块前面有c，就意味着它是用c语言重写了，也意味着速度更快些）。所以，在下面的操作中，不管是用import pickle，还是用import cpickle as pickle，在功能上都是一样的。

```
>>> import pickle
>>> integers = [1, 2, 3, 4, 5]
>>> f = open("22901.dat", "wb")
>>> pickle.dump(integers, f)
>>> f.close()
```

用pickle.dump（integers，f）将数据integers保存到文件22901.dat中。如果你要打开这个文件看里面的内容，可能会有点失望，但是，它对计算机是友好的。这个步骤可以称之为将对象序列化。用到的方法是：pickle.dump（obj，file[，protocol]）。

- obj：序列化对象，在上面的例子中是一个列表，它是基本类型，也可以序列化自己定义的类型。
- file：要写入的文件。可以更广泛地理解为拥有write()方法的对象，并且能接受字符串为参数，所以，它还可以是一个StringIO对象，或者其他自定义满足条件的对象。
- protocol：可选项。默认为False（或者说0），以ASCII格式保存对象；如果设置为1或者True，则以压缩的二进制格式保存对象。

换一种数据格式，并且做对比：

```
>>> import pickle
>>> d = {}
>>> integers = range(9999)
>>> d["i"] = integers                        #下面将这个dict格式的对象存入文

>>> f = open("22902.dat", "wb")
>>> pickle.dump(d, f)                        #文件中以ascii格式保存数据
>>> f.close()
```

```
>>> f = open("22903.dat", "wb")
>>> pickle.dump(d, f, True)            #文件中以二进制格式保存数据
>>> f.close()

>>> import os
>>> s1 = os.stat("22902.dat").st_size   #得到两个文件的大小
>>> s2 = os.stat("22903.dat").st_size

>>> print "%d, %d, %.2f%%" % (s1, s2, (s2+0.0)/s1*100)
68903, 29774, 43.21%
```

比较结果发现，以二进制方式保存的文件比以ASCII格式保存的文件小很多，前者约是后者的43%。

所以，在序列化的时候，特别是面对较大对象时，建议将dump()的参数True设置上，虽然现在存储设备的价格便宜，但是能省的还是省点比较好。

存入文件，还有另外一个目标，就是要读出来，也称之为反序列化。

```
>>> integers = pickle.load(open("22901.dat", "rb"))
>>> print integers
[1, 2, 3, 4, 5]
```

再看看被以二进制方式存入的那个文件：

```
>>> f = open("22903.dat", "rb")
>>> d = pickle.load(f)
>>> print d
{'i': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ....   #省略后
>>> f.close()
```

如果是自己定义的数据类型，是否可以用上述方式存入文件并读出来呢？看下面的例子：

```
>>> import cPickle as pickle          #cPickle更快
>>> import StringIO                   #标准库中的一个模块，跟file功能类
                                               #只不过是在内存中操作“
>>> class Book(object):               #自定义一种类型
...     def __init__(self,name):
...         self.name = name
...     def my_book(self):
...         print "my book is: ", self.name
...
```

```
>>> pybook = Book("<from beginner to master>")
>>> pybook.my_book()
my book is:  <from beginner to master>

>>> file = StringIO.StringIO()
>>> pickle.dump(pybook, file, 1)
>>> print file.getvalue()                 #查看“文件”内容，注意下面不是乱码
ccopy_reg
_reconstructor
q?(c__main__
Book
q?c__builtin__
object
q?NtRq?}qU?nameq?U?<from beginner to master>sb.

>>> pickle.dump(pybook, file)             #换一种方式，再看内容，可以比较一
>>> print file.getvalue()                 #视觉上两者就有很大差异
ccopy_reg
_reconstructor
q?(c__main__
Book
q?c__builtin__
object
q?NtRq?}qU?nameq?U?<from beginner to master>sb.ccopy_reg
_reconstructor
p1
(c__main__
Book
p2
c__builtin__
object
p3
NtRp4
(dp5
S'name'
p6
S'<from beginner to master>'
p7
sb.
```

如果从文件中读出来：

```
>>> file.seek(0)                         #找到对应类型
>>> pybook2 = pickle.load(file)
>>> pybook2.my_book()
my book is:  <from beginner to master>
>>> file.close()
```

7.2 shelve

pickle模块已经表现出它足够好的一面了。不过，由于数据的复杂性，pickle只能完成一部分工作，在另外更复杂的情况下，它就稍显麻烦了，于是，又有了shelve。

shelve模块也是标准库中的，先看看基本的写、读操作。

```
>>> import shelve
>>> s = shelve.open("22901.db")
>>> s["name"] = "www.itdiffer.com"
>>> s["lang"] = "python"
>>> s["pages"] = 1000
>>> s["contents"] = {"first":"base knowledge", "second":"day day u
>>> s.close()
```

以上完成了数据写入的过程，其实，这很接近数据库的样式了。下面是读。

```
>>> s = shelve.open("22901.db")
>>> name = s["name"]
>>> print name
www.itdiffer.com
>>> contents = s["contents"]
>>> print contents
{'second': 'day day up', 'first': 'base knowledge'}
```

看到输出的内容，你一定想到，肯定可以用for语句来读，想到了就用代码来测试，这就是Python交互模式的便利之处。

```
>>> for k in s:
...     print k, s[k]
...
contents {'second': 'day day up', 'first': 'base knowledge'}
lang python
pages 1000
name www.itdiffer.com
```

不管是写还是读，都似乎要简化了。所建立的对象s，就如同字典一样，可称之为类字典对象，所以，可以如同操作字典那样来操作

它。

但是，小心有坑：

```
>>> f = shelve.open("22901.db")
>>> f["author"]
['qiwsir']
>>> f["author"].append("Hetz")     #试图增加一个
>>> f["author"]                    #坑就在这里
['qiwsir']
>>> f.close()
```

当试图修改一个已有键的值时没有报错，但是并没有修改成功。要填平这个坑，需要这样做：

```
>>> f = shelve.open("22901.db", writeback=True)    #多一个参数True
>>> f["author"].append("Hetz")
>>> f["author"]                                   #没有坑了
['qiwsir', 'Hetz']
>>> f.close()
```

还用for循环一下：

```
>>> f = shelve.open("22901.db")
>>> for k,v in f.items():
...     print k,": ",v
...
contents :  {'second': 'day day up', 'first': 'base knowledge'}
lang :  python
pages :  1000
author :  ['qiwsir', 'Hetz']
name :  www.itdiffer.com
```

shelve更像数据库了。不过，它还不是真正的数据库，真正的数据库在后面。

7.3 MySQL数据库

尽管用文件形式将数据保存到磁盘，已经是一种不错的方式。但是，人们还是发明了更具有格式化特点，并且写入和读取更快速便捷的东西——数据库。维基百科对数据库有比较详细的说明：

数据库指的是以一定方式储存在一起、能为多个用户共享、具有尽可能小的冗余度、与应用程序彼此独立的数据集合。

到目前为止，地球上有以下三种类型的数据。

- 关系型数据库：MySQL、Microsoft Access、SQL Server、Oracle……
- 非关系型数据库：MongoDB、BigTable（Google）……
- 键值数据库：Apache Cassandra（Facebook）、LevelDB（Google）……

7.3.1 MySQL概况

MySQL是一个使用非常广泛的数据库，很多网站都使用它。关于这个数据库有很多传说，例如维基百科上有这么一段：

MySQL原本是一个开放源代码的关系数据库管理系统，原开发者为瑞典的MySQL AB公司，该公司于2008年被太阳微系统（Sun Microsystems）收购。2009年，甲骨文公司（Oracle）收购太阳微系统公司，MySQL成为Oracle旗下产品。

MySQL在过去由于性能高、成本低、可靠性好，已经成为最流行的开源数据库，因此被广泛地应用在Internet上的中小型网站中。随着MySQL的不断成熟，也逐渐被用于更多大规模网站和应用，比如维基百科、Google和Facebook等网站。非常流行的开源软件组合LAMP中的“M”指的就是MySQL。

但被甲骨文公司收购后，Oracle大幅调涨MySQL商业版的售价，且甲骨文公司不再支持另一个自由软件项目OpenSolaris的发展，因此导致自由软件社区们对于Oracle是否还会持续支持MySQL社区版

（MySQL之中唯一的免费版本）有所隐忧，因此原先一些使用MySQL的开源软件逐渐转向其他的数据库。例如维基百科已于2013年正式宣布将从MySQL迁移到MariaDB数据库。

不管怎样，MySQL依然是一个不错的数据库选择，足够支持读者完成一个不小的网站。

7.3.2 安装

你的电脑或许不会天生就有MySQL，它本质上也是一个程序，若有必要，需安装。

我用Ubuntu操作系统演示，因为我相信读者将来在真正的工程项目中，多数情况下要操作Linux系统的服务器，并且，我酷爱用Ubuntu。本书的目标是from beginner to master，不管是不是真的master，也要装得像，Linux能够给你撑门面，这也是推荐使用Ubuntu的原因。

第一步，在shell端运行如下命令：

```
sudo apt-get install mysql-server
```

运行完毕就安装好了这个数据库，是不是很简单呢？当然，还要进行配置。

第二步，配置MySQL

安装之后，运行：

```
service mysqld start
```

启动MySQL数据库，然后进行下面的操作，对其进行配置。

默认的MySQL安装之后根用户是没有密码的，注意，这里有一个名词“根用户”，其用户名是：root。运行：

```
$mysql -u root
```

进入MySQL之后，会看到“>”符号开头，这就是MySQL的命令操

作界面了。

下面设置MySQL中的root用户密码，否则，MySQL服务无安全可言了。

```
mysql> GRANT ALL PRIVILEGES ON *.* TO root@localhost IDENTIFIED BY
```

用123456作为root用户的密码，应该是非常愚蠢的，在真正的项目中最好别这样做，要用大小写字母与数字混合的密码，且不少于8位。以后如果再登录数据库，就可以用刚才设置的密码了。

7.3.3 运行

安装完就要运行它，并操作这个数据库。

```
$ mysql -u root -p
Enter password:
```

输入数据库的密码，之后出现：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 373
Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current inpu

mysql>
```

恭喜你，已经进入到数据操作界面了，接下来就可以对这个数据进行操作了。例如：

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
```

```
| carstore           |
| cutvideo           |
| itdiffer           |
| mysql              |
| performance_schema |
| test               |
+--------------------+
```

“show databases;”（最后的半角分号别忘记了）命令表示列出当前已经有的数据库。

对数据库的操作，除了用命令之外，还可以使用一些可视化工具，比如phpmyadmin就是不错的。

更多数据库操作的知识，这里就不再介绍了，读者可以参考有关书籍。

MySQL数据库已经安装好，但是Python还不能操作它，还要继续安装Python操作数据库的模块——python-MySQLdb

7.3.4 安装python-MySQLdb

python-MySQLdb是一个接口程序，Python通过它对MySQL数据实现各种操作。

在编程中会遇到很多类似的接口程序，通过接口程序对另外一个对象进行操作。接口程序就好比钥匙，如果要开锁，直接用手指去捅肯定是不行的，必须借助工具插入到锁孔中，把锁打开，门开了，就可以操作门里面的东西了，那么打开锁的工具就是接口程序。谁都知道，用对应的钥匙开锁是最好的，如果用别的工具（比如锤子）或许不便利（当然，具有特殊开锁能力的人除外），也就是接口程序，编码水平等都是考虑因素。

python-MySQLdb就是打开MySQL数据库的钥匙。

如果要源码安装，其源码下载地址：https://pypi.python.org/pypi/MySQL-python/。下载之后就可以安装了。

在Ubuntu操作系统下还可以用软件仓库来安装。

```
sudo apt-get install build-essential python-dev libmysqlclient-dev
```

```
sudo apt-get install python-MySQLdb
```

也可以用pip来安装：

```
pip install mysql-python
```

安装之后，在Python交互模式下：

```
>>> import MySQLdb
```

如果不报错，那么恭喜你，已经安装好了。如果报错，那么恭喜你，可以借着错误信息提高自己的计算机水平。

7.3.5 连接数据库

连接数据库之前要先建立数据库。

```
$ mysql -u root -p
Enter password:
```

进入到数据库操作界面：

```
mysql>
```

输入如下命令，建立一个数据库：

```
mysql> create database qiwsirtest character set utf8;
Query OK, 1 row affected (0.00 sec)
```

注意上面的指令，如果仅仅输入create database qiwsirtest也可以，但是我在后面增加了character set utf8，意思是所建立的数据库qiwsirtest，编码是utf-8，这样存入汉字就不是乱码了。

看到那一行提示：Query OK，1 row affected（0.00 sec），说明这个数据库已经建立好了，名字叫作qiwsirtest

数据库建立之后，就可以用Python通过已经安装的python-MySQLdb来连接这个名字叫作qiwsirtest的库了。

```

```

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(host="localhost", user="root", passwd="
```

下面逐个解释上述命令的含义。

- host：等号的后面应该填写MySQL数据库的地址，因为数据库就在本机上，所以使用localhost，注意引号。如果在其他的服务器上，这里应该填写IP地址。一般中小型的网站，数据库和程序都是在同一台服务器（计算机）上，就使用localhost了。
- user：登录数据库的用户名，这里一般填写“root”，还是要注意引号。当然，如果读者命名了别的用户名，就更改为相应用户。但是，不同用户的权限可能不同，所以，在程序中，如果要操作数据库，还要注意所拥有的权限。在这里用root，就什么权限都有了。不过，这种做法在大型系统中是应该避免的。
- passwd：user账户登录MySQL的密码。例子中用的密码是“123123”，不要忘记引号。
- db：就是刚刚通create命令建立的数据库，数据库名字是“qiwsirtest”，还是要注意引号。如果你建立的数据库名字不是这个，就写自己所建数据库的名字。
- port：一般情况，MySQL的默认端口是3306。当MySQL被安装到服务器之后，为了能够允许网络访问，服务器（计算机）要提供一个访问端口给它（服务器管理员可以进行配置端口）。
- charset：这个设置，在很多教程中都不写，结果在真正进行数据存储的时候，发现有乱码。这里将qiwsirtest这个数据库的编码设置为utf-8格式，这样就允许存入汉字而无乱码了。注意，在MySQL设置中，utf-8写成utf8，没有中间的横线，但是在Python文件开头和其他地方设置编码格式的时候要写成utf-8。

注：connect中的所有参数，可以只按照顺序把值写入。但是，我推荐读者的写法还是上面的方式，以免出了乱子自己还糊涂。

其实，关于connect的参数还有别的，读者可以到MySQLdb官方查看文档，此处就不再赘述。特别提醒，官方文档是最好的教材，最值得反复阅读。

至此，已经完成了数据库的连接。

7.3.6　数据库表

就数据库而言，连接之后就要对其操作。但是，目前名字叫作qiwsirtest的数据库仅仅是空架子，没有什么可操作的，要操作它，就必须在里面建立“表”，什么是数据库的表呢？下面摘抄维基百科对数据库表的简要解释。

在关系数据库中，数据库表是一系列二维数组的集合，用来代表和储存数据对象之间的关系。它由纵向的列和横向的行组成，例如一个有关作者信息的名为authors的表中，每个列包含的是所有作者的某个特定类型的信息，比如“姓氏”，而每行则包含了某个特定作者的所有信息：姓、名、住址等。

对于特定的数据库表，列的数目一般事先固定，各列之间可以由列名来识别。而行的数目可以随时动态变化，通常每行都可以根据某个（或某几个）列中的数据来识别，称为候选键。

在qiwsirtest中建立一个存储用户名、用户密码、用户邮箱的表，其结构用二维表格表现如下：

username	password	email
qiwsir	123123	qiwsir@gmail.com

特别说明，这里为了简化细节、突出重点，对密码不加密，直接明文保存，虽然这种方式是很不安全的。据小道消息，有的网站居然用明文保存密码，这么做的目的是比较可恶的。就让我在这里，仅仅在这里可恶一次。

因为直接操作数据不是本书重点，但是关联到后面的操作，为了让读者在阅读上连贯，快速地说明建立数据库表并输入内容。

```
mysql> use qiwsirtest;
Database changed
mysql> show tables;
Empty set (0.00 sec)
```

用show tables命令显示这个数据库中是否有数据表了，查询结果显示为空。

用如下命令建立一个数据表，这个数据表的内容就是上面所说明的。

mysql＞create table users(id int(2) not null primary key auto_increment,username varchar(40),password text,email

text)default charset=utf8;

```
Query OK, 0 rows affected (0.12 sec)
```

建立的这个数据表名称是：users，其中包含上述字段，可以用下面的方式看一看这个数据表的结构。

```
mysql> show tables;
+----------------------+
| Tables_in_qiwsirtest |
+----------------------+
| users                |
+----------------------+
1 row in set (0.00 sec)
```

查询显示，在qiwsirtest这个数据库中已经有一个表，它的名字是：users。

```
mysql> desc users;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | int(2)      | NO   | PRI | NULL    | auto_increment |
| username | varchar(40) | YES  |     | NULL    |                |
| password | text        | YES  |     | NULL    |                |
| email    | text        | YES  |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

显示出来表users的结构。

特别提醒：上述所有字段设置仅为演示，在实际开发中，要根据具体情况来确定字段的属性。

如此就得到了一个空表，可以查询看看：

```
mysql> select * from users;
Empty set (0.01 sec)
```

向里面插入一条信息：

```
mysql> insert into users(username,password,email)
values("qiwsir","123123", "qiwsir@gmail.com");
Query OK, 1 row affected (0.05 sec)
mysql> select * from users;
```

```
+----+----------+----------+------------------+
| id | username | password | email            |
+----+----------+----------+------------------+
|  1 | qiwsir   | 123123   | qiwsir@gmail.com |
+----+----------+----------+------------------+
1 row in set (0.00 sec)
```

这样就得到了一个有内容的数据库表。

7.3.7 操作数据库

连接数据库。

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(host="localhost",user="root",passwd="12
```

Python建立了与数据的连接，其实是建立了一个MySQLdb.connect()的实例对象，或者泛泛地称之为连接对象，Python就是通过连接对象和数据库对话。这个对象常用的方法如下。

- commit()：如果数据库表进行了修改，提交保存当前的数据。当然，如果此用户没有权限就作罢，什么也不会发生。
- rollback()：如果有权限，就取消当前的操作，否则报错。
- cursor（[cursorclass]）：返回连接的游标对象。通过游标执行SQL查询并检查结果。游标比连接支持更多的方法，而且可能在程序中更好用。
- close()：关闭连接。此后，连接对象和游标都不再可用了。

Python和数据之间的连接建立起来之后，若要操作数据库，就需要让Python对数据库执行SQL语句。

Python是通过游标执行SQL语句的，所以，连接建立之后，就要利用连接对象得到游标对象，方法如下：

```
>>> cur = conn.cursor()
```

此后，就可以利用游标对象的方法对数据库进行操作，那么还得了解游标对象的常用方法，如表7-1所示。

表7-1　游标对象的常用方法

名　称	描　述
close()	关闭游标
execute(query[,args])	执行一条 SQL 语句，可以带参数
executemany(query, pseq)	对序列 pseq 中的每个参数执行 sql 语句
fetchone()	返回一条查询结果
fetchall()	返回所有查询结果
fetchmany([size])	返回 size 条结果
nextset()	移动到下一个结果
scroll(value,mode='relative')	移动游标到指定行，mode='relative'，表示从当前行开始移动 value 条；mode='absolute'，表示从第一行开始移动 value 条

1.插入

例如，要在数据表users中插入一条记录，使得：username="python"，password="123456"，email="python@gmail.com"，这样做：

```
>>> cur.execute("insert into users (username,password,email) value
1L
```

没有报错，并且返回一个"1L"结果，说明有一行记录操作成功。不妨进入到“mysql>”交互方式查看（读者可以在另外一个shell中进行操作）：

```
mysql> select * from users;
```

```
+----+----------+----------+------------------+
| id | username | password | email            |
+----+----------+----------+------------------+
|  1 | qiwsir   | 123123   | qiwsir@gmail.com |
+----+----------+----------+------------------+
1 row in set (0.00 sec)
```

怎么没有看到增加的那一条呢？哪里错了？上面也没有报错。

特别注意，通过“cur.execute()”对数据库进行操作之后，没有报错，完全正确，但是不等于数据就已经提交到数据库中了，还必须要用到“MySQLdb.connect”的一个属性：commit()，将数据提交上去，也就是进行了“cur.execute()”操作之后，必须要将数据提交才能有效改变数据库表。

```
>>> conn.commit()
```

再到“mysql＞”中运行“select*from users”试一试：

```
mysql> select * from users;
```

```
+----+----------+----------+------------------+
| id | username | password | email            |
+----+----------+----------+------------------+
|  1 | qiwsir   | 123123   | qiwsir@gmail.com |
|  2 | python   | 123456   | python@gmail.com |
+----+----------+----------+------------------+
2 rows in set (0.00 sec)
```

果然如此。

这就如同编写一个文本一样，将文字写到文本上，并不等于文字已经保留在文本文件中了，必须执行“CTRL-S”才能保存。所有以“execute()”执行的各种sql语句之后，要让已经执行的效果保存，都必须运行连接对象的“commit()”方法。

再尝试一下插入多条的那个命令“executemany（query，args）”。

```
>>> cur.executemany("insert into users (username,password,email) v
4L
>>> conn.commit()
```

到“mysql＞”里面看结果：

```
mysql> select * from users;
```

```
+----+----------+----------+------------------+
| id | username | password | email            |
+----+----------+----------+------------------+
|  1 | qiwsir   | 123123   | qiwsir@gmail.com |
|  2 | python   | 123456   | python@gmail.com |
|  3 | google   | 111222   | g@gmail.com      |
|  4 | facebook | 222333   | f@face.book      |
|  5 | github   | 333444   | git@hub.com      |
|  6 | docker   | 444555   | doc@ker.com      |
+----+----------+----------+------------------+
6 rows in set (0.00 sec)
```

成功插入了多条记录。在“executemany（query，pseq）”中，query还是一条sql语句，但是pseq这时候是一个元组，特别注意括号——一环套一环的括号，这个元组里面的元素也是元组，每个元组分别对应sql语句中的字段列表。

除了插入命令，其他对数据操作的命令都可以用类似上面的方式，比如删除、修改等。

2.查询

如果要从数据库中查询数据，也用游标方法来操作。

```
>>> cur.execute("select * from users")
7L
```

说明从users表汇总查询出来了7条记录。但是，这似乎有点不友好，7条记录在哪里呢？如果在“mysql＞”下操作查询命令，一下子就把7条记录列出来了，在这里怎么显示Python的查询结果呢？

要用到游标对象的fetchall()、fetchmany（size＝None）、fetchone()、scroll（value，mode＝'relative'）等方法。

```
>>> cur.execute("select * from users")
7L
>>> lines = cur.fetchall()
```

至此已经将查询到的记录赋值给变量lines了，如果要把它们显示出来，就要用到曾经学习过的循环语句。

```
>>> for line in lines:
...     print line
...
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
(2L, u'python', u'123456', u'python@gmail.com')
(3L, u'google', u'111222', u'g@gmail.com')
(4L, u'facebook', u'222333', u'f@face.book')
(5L, u'github', u'333444', u'git@hub.com')
(6L, u'docker', u'444555', u'doc@ker.com')
(7L, u'\u8001\u9f50', u'9988', u'qiwsir@gmail.com')
```

很好，果然逐条显示出来了。请读者注意，第七条中的u'\u8001\u95f5'，在这里是汉字，只是由于我的shell不能显示罢了，

不必惊慌，也不必搭理它。

只想查出第一条，可以吗？当然可以，再看下面：

```
>>> cur.execute("select * from users where id=1")
1L
>>> line_first = cur.fetchone()                    #只返回一条
>>> print line_first
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
```

为了对上述过程了解深入，做下面的实验：

```
>>> cur.execute("select * from users")
7L
>>> print cur.fetchall()
((1L, u'qiwsir', u'123123', u'qiwsir@gmail.com'), (2L, u'python',
```

原来，用cur.execute()从数据库查询出来的东西，被“保存在了cur所能找到的某个地方”，要找出这些被保存的东西，需要用cur.fetchall()（或者fechone等），并且找出来之后，作为对象存在。从上面的实验探讨发现，返回值是一个元组对象，里面的每个元素，又是一个一个的元组对象，因此，用for循环就可以一个一个拿出来了。

接着上面的操作，再打印一遍。

```
>>> print cur.fetchall()
()
```

怎么是空？不是说作为对象已经存在于内存中了吗？难道这个内存中的对象仅一次有效吗？

不要着急，这就是神奇所在。

通过游标找出来的对象，在读取的时候有一个特点，就是那个游标会移动。在第一次操作了print cur.fetchall()后，因为是将所有的都打印出来，游标就从第一条移动到最后一条。当print结束之后，游标已经在最后一条的后面了。接下来如果再次打印，就空了，最后一条后面没有东西了。

下面还要进行实验，检验上面所说：

```
>>> cur.execute('select * from users')
7L
>>> print cur.fetchone()
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
>>> print cur.fetchone()
(2L, u'python', u'123456', u'python@gmail.com')
>>> print cur.fetchone()
(3L, u'google', u'111222', u'g@gmail.com')
```

这次不再一次性全部打印出来了，而是每次打印一条，从结果中可以看出来，那个游标果然在一条一条向下移动呢。注意，在这次实验中重新运行了查询语句。

那么，既然操作存储在内存中的对象时游标会移动，能不能让游标向上移动，或者移动到指定位置呢？这就是scroll()。

```
>>> cur.scroll(1)
>>> print cur.fetchone()
(5L, u'github', u'333444', u'git@hub.com')
>>> cur.scroll(-2)
>>> print cur.fetchone()
(4L, u'facebook', u'222333', u'f@face.book')
```

果然，这个函数能够移动游标，请仔细观察，上面的方式是让游标相对于当前位置向上或者向下移动。即cur.scroll（n）或者cur.scroll（n，"relative"），意思是相对于当前位置向上或者向下移动，若n为正数，则表示向下（向前），若n为负数，则表示向上（向后）

还有一种方式可以实现“绝对”移动，而不是“相对”移动：增加一个参数"absolute"。

但在Python中，序列对象的顺序是从0开始的。

```
>>> cur.scroll(2, "absolute")                    #回到序号是2，但指向第三条
>>> print cur.fetchone()                         #打印，果然是
(3L, u'google', u'111222', u'g@gmail.com')

>>> cur.scroll(1, "absolute")
>>> print cur.fetchone()
(2L, u'python', u'123456', u'python@gmail.com')

>>> cur.scroll(0, "absolute")                    #回到序号是0,即指向第一条
>>> print cur.fetchone()
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
```

至此，已经熟悉了cur.fetchall()和cur.fetchone()以及cur.scroll()的几个方法，还有一个方法，即游标在序号是1的位置，指向第二条。

```
>>> cur.fetchmany(3)
((2L, u'python', u'123456', u'python@gmail.com'), (3L, u'google',
```

上面这个操作，就是实现了从当前位置（游标指向tuple序号为1的位置，即第二条记录）开始，含当前位置，向下列出3条记录。

读取数据，好像有点啰嗦，但细细琢磨，还是有道理的。

Python总是能够为我们着想的，在连接对象的游标方法中提供了一个参数，可以实现将读取到的数据变成字典形式，这样就提供了另外一种读取方式。

```
>>> cur = conn.cursor(cursorclass=MySQLdb.cursors.DictCursor)
>>> cur.execute("select * from users")
7L
>>> cur.fetchall()
({'username': u'qiwsir', 'password': u'123123', 'id': 1L, 'email':
```

这样，在元组里面的元素就是一个一个字典：

```
>>> cur.scroll(0,"absolute")
>>> for line in cur.fetchall():
...     print line["username"]
...
qiwsir
mypython
google
facebook
github
docker
```

根据字典对象的特点来读取“键-值”。

7.3.8　更新数据

熟悉了前面的操作，再到这里一切都显得那么简单。但仍要提醒的是，如果更新完毕，和插入数据一样，都需要commit()来提交保

存。

```
>>> cur.execute("update users set username=%s where id=2",("mypyth
1L
>>> cur.execute("select * from users where id=2")
1L
>>> cur.fetchone()
(2L, u'mypython', u'123456', u'python@gmail.com')
```

从操作中可以看出，已经将数据库中第二条的用户名修改为mypython了，用的就是update语句。

不过，要真的实现在数据库中的更新，还要运行：

```
>>> conn.commit()
```

还有个小尾巴，即当你操作数据完毕，不要忘记关门：

```
>>> cur.close()
>>> conn.close()
```

门锁好了，放心离开。

7.4 MongoDB数据库

MongoDB开始火了，这是时代发展的需要。为此，在这里也探讨一下如何用Python来操作此数据库。考虑到读者对这种数据库的了解可能比关系型数据库陌生，所以，要用多一点的篇幅来介绍。

mongodb是属于NoSql的。

NoSql（Not Only Sql）指的是非关系型的数据库。它是为了大规模Web应用而生的，其特征诸如模式自由、支持简易复制、简单的API、大容量数据等。

MongoDB是NoSql之一，选择它，主要是因为我喜欢，下面说说它的特点：

- 面向文档存储
- 对任何属性可索引
- 复制和高可用性
- 自动分片
- 丰富的查询
- 快速就地更新

基于它的特点，擅长的领域在于：

- 大数据（太时髦了！以下都可以不看，有这么一条就足够了）
- 内容管理和交付
- 移动和社交基础设施
- 用户数据管理
- 数据平台

7.4.1 安装MongoDB

先演示在Ubuntu系统中的安装过程：

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
```

```
sudo apt-get update
sudo apt-get install mongodb-10gen
```

如此就安装完了。安装流程可以参考：http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/。

如果你用的是其他操作系统，可以到官方网站下载安装程序：http://www.mongodb.org/downloads，该网站能满足各种操作系统。

如果在安装过程中遇到了问题，建议去问Google大神（如果有读者心存疑虑或者愤愤不平，请不要发怒，这是我的个人建议，不同意可以略过，我当然也尊重读者的个人选择）。

7.4.2　启动

安装完毕就可以启动数据库了。因为本书不是专门讲数据库的，所以这里不涉及数据库的详细讲解，下面只是建立一个简单的库，并且说明MongoDB的基本要点，目的在于为后面用Python来操作它做个铺垫。

执行mongo启动shell，显示的也是“>”，有点类似mysql的状态。在shell中，可以实现与数据库的交互操作。

在shell中，有一个全局变量db，使用哪个数据库，哪个数据库就会被复制给这个全局变量db，如果那个数据库不存在，就会新建。

```
> use mydb
switched to db mydb
> db
mydb
```

除非向这个数据库中增加实质性的内容，否则它是看不到的。

```
> show dbs;
local   0.03125GB
```

向这个数据库增加点东西。MongoDB的基本单元是文档，所谓文档，类似于Python中的字典，以键值对的方式保存数据。

```
> book = {"title":"from beginner to master", "author":"qiwsir", "l
{
    "title" : "from beginner to master",
    "author" : "qiwsir",
    "lang" : "python"
}
> db.books.insert(book)
> db.books.find()
{ "_id" : ObjectId("554f0e3cf579bc0767db9edf"), "title" : "from be
```

db指向了数据库mydb，books是这个数据库里面的一个集合（类似mysql里面的表），向集合books里面插入了一个文档（文档对应mysql里面的记录）。“数据库、集合、文档”构成了mongodb数据库。

从上面的操作还发现一个有意思的地方，并没有类似create之类的命令，用到数据库，就通过use xxx操作，如果不存在就建立；用到集合，就通过db.xxx来使用，如果没有就建立。可以总结为“随用随取随建立”。是不是简单的有点出人意料？

```
> show dbs
local    0.03125GB
mydb     0.0625GB
```

当有了充实内容之后，会看到刚才用到的数据库mydb。

在shell中，可以对数据施以“增删改查”等操作。但是，我们的目的是用Python来操作，所以，还是把力气放在后面用。

7.4.3　安装pymongo

要用Python来驱动MongoDB，必须要安装驱动模块，即pymongo，这跟操作mysql类似。安装方法推荐如下：

```
$ sudo pip install pymongo
```

如果顺利，就会看到最后的提示：

```
Successfully installed pymongo
Cleaning up...
```

在写本书的时候，安装版本号如下，如果读者的版本不一样，也无大碍。

```
>>> import pymongo
>>> pymongo.version
'3.0.1'
```

如果读者要指定版本，比如安装2.8版本的，可以：

```
$ sudo pip install pymongo==2.8
```

安装好之后，进入到Python的交互模式：

```
>>> import pymongo
```

说明模块没有问题。

7.4.4 连接MongoDB

既然Python驱动MongDB的模块pymongo业已安装完毕，接下来就是连接，即建立连接对象。

```
>>> pymongo.Connection("localhost",27017)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'Connection'
```

报错！我在写这本书之前做项目时，就是按照上面的方法链接的，读者可以查一下，会发现很多教程都是这么连接的。但是，眼睁睁地看到了报错。

所以，一定要注意这里的坑。

如果读者用的是旧版本的pymongo，比如2.8版本，仍然可以使用上面的连接方法，如果是像我一样用的新版本，就得注意这个问题了。

经验主义害死人。必须看看下面有哪些方法可以用：

```
>>> dir(pymongo)
['ALL', 'ASCENDING', 'CursorType', 'DESCENDING', 'DeleteMany', 'De
```

瞪大我的那双浑浊迷茫、布满血丝、渴望惊喜的眼睛，透过近视镜的玻璃片，怎么也找不到Connection()这个方法。原来，刚刚安装的pymongo变了，“他变了”。

不过，我发现了MongoClient()，这是一个峰回路转的发现。

```
>>> client = pymongo.MongoClient("localhost", 27017)
```

很好，Python已经和MongoDB建立了连接。

刚才已经建立了一个数据库mydb，并且在这个库里面有一个集合books，于是：

```
>>> db = client.mydb
```

或者：

```
>>> db = client['mydb']
```

获得数据库mydb，并赋值给变量db（这个变量不是MongoDB的shell中那个db，此处的db就是Python中一个寻常的变量）。

```
>>> db.collection_names()
[u'system.indexes', u'books']
```

查看集合，发现了已经建立好的那个books，于是再获取这个集合，并赋值给一个变量books：

```
>>> books = db["books"]
```

或者：

```
>>> books = db.books
```

接下来，就可以操作这个集合中的具体内容了。

7.4.5 编辑

刚刚的books所引用的是一个MongoDB的集合对象，它跟前面学习过的其他对象一样，有一些方法供我们来驱使。

```
>>> type(books)
<class 'pymongo.collection.Collection'>

>>> dir(books)
['_BaseObject__codec_options', '_BaseObject__read_preference', '_B
```

这么多方法不会一一介绍，只是按照“增删改查”的常用功能介绍几种。读者可以使用help()去查看每一种方法的使用说明。

```
>>> books.find_one()
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'),
```

提醒读者注意的是，MongoDB的shell中的命令与pymongo中的方法有时候会稍有差别，应务必小心。比如刚才这个，在shell中是这样子的：

```
> db.books.findOne()
{
    "_id" : ObjectId("554f0e3cf579bc0767db9edf"),
    "title" : "from beginner to master",
    "author" : "qiwsir",
    "lang" : "python"
}
```

请注意区分。

目前在集合books中有一个文档，还想再增加，于是就进入到了“增删改查”的常规操作。

1.新增和查询

```
>>> b2 = {"title":"physics", "author":"Newton", "lang":"english"}
>>> books.insert(b2)
ObjectId('554f28f465db941152e6df8b')
```

成功地向集合中增加了一个文档。得看看结果（我们就是充满好

奇心的小孩子，记得女儿小时候，每次给她照相，每拍一张，她总要看一看。看看就是一种查询。

```
>>> books.find().count()
2
```

这是查看当前集合有多少个文档的方式，返回值为2，则说明有两条文档了。还是要看看内容。

```
>>> books.find_one()
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'),
```

这个命令就不行了，因为它只返回第一条。必须要：

```
>>> for i in books.find():
...     print i
...
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'),
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId ('554
```

在books引用的对象中有find()方法，它返回的是一个可迭代对象，包含着集合中所有的文档。

由于文档是“键/值”对，不一定每条文档都要结构一样，比如，也可以在集合中插入这样的文档。

```
>>> books.insert({"name":"Hertz"})
ObjectId('554f2b4565db941152e6df8c')
>>> for i in books.find():
...     print i
...
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'),
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId ('554
{u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz'}
```

如果有多个文档，想一下子都插入到集合中（在MySQL中，可以实现多条数据用一条命令插入到表里面），可以这么做：

```
>>> n1 = {"title":"java", "name":"Bush"}
>>> n2 = {"title":"fortran", "name":"John Warner Backus"}
>>> n3 = {"title":"lisp", "name":"John McCarthy"}
>>> n = [n1, n2, n3]
>>> n
```

```
[{'name': 'Bush', 'title': 'java'}, {'name': 'John Warner Backus',
>>> books.insert(n)
[ObjectId('554f30be65db941152e6df8d'), ObjectId('554f30be65db94115
```

这样就完成了所谓的批量插入，查看一下文档条数：

```
>>> books.find().count()
6
```

要提醒读者，批量插入的文档的大小是有限制的，有人说不要超过20万条，也有人说不要超过16MB，但我没有测试过。在一般情况下，或许达不到上限，如果遇到极端情况，就请读者在使用时多注意。

如果要查询，除了通过循环之外，能不能按照某个条件查呢？比如查找'name'='Bush'的文档：

```
>>> books.find_one({"name":"Bush"})
{u'_id': ObjectId('554f30be65db941152e6df8d'), u'name': u'Bush', u
```

对于查询结果，还可以进行排序：

```
>>> for i in books.find().sort("title", pymongo.ASCENDING):
...     print i
...
{u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz'}
{u'_id': ObjectId('554f30be65db941152e6df8e'), u'name': u'John War
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'),
{u'_id': ObjectId('554f30be65db941152e6df8d'), u'name': u'Bush', u
{u'_id': ObjectId('554f30be65db941152e6df8f'), u'name': u'John McC
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId ('554
```

这是按照"title"的值的升序排列的，注意sort()中的第二个参数，意思是升序排列。如果按照降序，就需要将参数修改为pymongo.DESCEDING，也可以指定多个排序键。

```
>>> for i in
books.find().sort([("name",pymongo.ASCENDING),("name",pymongo.DESC
...     print i
...
{u'_id': ObjectId('554f30be65db941152e6df8e'), u'name': u'John War
{u'_id': ObjectId('554f30be65db941152e6df8f'), u'name': u'John McC
{u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz'}
{u'_id': ObjectId('554f30be65db941152e6df8d'), u'name': u'Bush', u
```

```
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'),
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId ('554
```

如果读者看到这里，请务必注意，MongoDB中的每个文档，本质上都是“键/值”对的类字典结构。这种结构，一经Python读出来，就可以用字典中的各种方法来操作。与此类似的还有一个名为json的东西，但是，用Python读过来之后，无法直接用json模块中的json.dumps()方法操作文档。其中一种解决方法就是将文档中的'_id'键/值对删除（例如：del doc['_id']），然后使用json.dumps()即可。读者也可使用json_util模块，因为它是“Tools for using Python’s json module with BSON documents”，请阅读http://api.mongodb.org/python/current/api/bson/json_util.html中的模块使用说明。

2.更新

对于已有数据，更新是数据库中常用的操作。比如，要更新name为Hertz那个文档：

```
>>> books.update({"name":"Hertz"}, {"$set": {"title":"new physics"
{u'updatedExisting': True, u'connectionId': 4, u'ok': 1.0, u'err':
>>> books.find_one({"author":"Hertz"})
{u'title': u'new physics', u'_id': ObjectId('554f2b4565db941152e6c
```

在更新的时候，用了一个$set修改器，它可以用来指定键值，如果键不存在，就会创建。

关于修改器，不仅仅是这一个，还有别的呢，如表7-1所示。

表7-1　修改器

修改器	描　述
$set	用来指定一个键的值。如果不存在则创建它
$unset	完全删除某个键
$inc	增加已有键的值，不存在则创建（只能用于增加整数、长整数、双精度浮点数）
$push	数组修改器只能操作值为数组，存在 key 则在值末尾增加一个元素，不存在则创建一个数组

3.删除

删除可以用remove()方法，稍一演示，读者必会。

```
>>> books.remove({"name":"Bush"})
```

```
{u'connectionId': 4, u'ok': 1.0, u'err': None, u'n': 1}
>>> books.find_one({"name":"Bush"})
>>>
```

这是将那个文档全部删除。当然，也可以根据MongoDB的语法规则写个条件，按照条件删除。

4.索引

索引的目的是为了让查询速度更快，当然，在具体的项目开发中，是否建立索引要视情况而定，因为建立索引也是有代价的。

```
>>> books.create_index([("title", pymongo.DESCENDING),])
u'title_-1'
```

这里仅仅是对pymongo模块做了一个非常简单的介绍，在实际使用过程中，上面的知识是很有限的，所以需要读者根据具体应用场景再结合MongoDB的有关知识去尝试新的语句。

7.5 SQLite数据库

SQLite是一个小型的关系型数据库，它最大的特点在于不需要服务器、零配置。前面的两个数据库，不管是MySQL还是MongoDB，都需要“安装”，安装之后，才运行起来，其实是已经有一个相应的服务器在跑着呢。而SQLite不需要这样，首先Python已经将相应的驱动模块作为标准库一部分了，只要安装了Python，就可以使用；另外，它也不需要服务器，可以类似操作文件那样来操作SQLite数据库文件。还有一点也不错，SQLite源代码不受版权限制。

SQLite也是一个关系型数据库，所以SQL语句也可以在里面使用。

与操作MySQL数据库类似，对于SQLite数据库也要通过以下几步。

- 建立连接对象。
- 连接对象方法：建立游标对象。
- 游标对象方法：执行sql语句。

7.5.1 建立连接对象

由于SQLite数据库的驱动已经在Python里面了，所以，只要引用就可以直接使用。并且在学过MySQL的基础上，理解本节内容就容易多了。

```
>>> import sqlite3
>>> conn = sqlite3.connect("23301.db")
```

这样就得到了连接对象，是不是比MySQL连接要简化了很多呢。在sqlite3.connect（"23301.db"）语句中，如果已经有了那个数据库，就连接上它；如果没有，就新建一个。注意，这里的路径可以随意指定。

不妨到目录中看一看，刚才建立的数据库文件是否存在了。

```

```

```
/2code$ ls 23301.db
23301.db
```

果然有了。连接对象建立起来之后，就要使用连接对象的方法继续工作了。

```
>>> dir(conn)
['DataError', 'DatabaseError', 'Error', 'IntegrityError', 'Interfa
```

7.5.2　游标对象

这一步跟MySQL也类似，要建立游标对象。

```
>>> cur = conn.cursor()
```

接下来对数据库内容进行操作，都是用游标对象方法来实现：

```
>>> dir(cur)
['__class__', '__delattr__', '__doc__', '__format__', '__getattrib
```

看到熟悉的名称了：close()、execute()、executemany()、fetchall()。

1.创建数据库表

对sqlite数据库，以往你熟悉的SQL语句照样可以使用，就如此操作一番。

```
>>> create_table = "create table books (title text, author text, l
>>> cur.execute(create_table)
<sqlite3.Cursor object at 0xb73ed5a0>
```

这样就在数据库23301.db中建立了一个表books，对这个表可以增加数据了。

```
>>> cur.execute('insert into books values ("from beginner to maste
<sqlite3.Cursor object at 0xb73ed5a0>
```

为了保证数据能够保存，还要如下操作（这是多么熟悉的操作流程和命令呀）：

```
>>> conn.commit()
>>> cur.close()
>>> conn.close()
```

在刚才建立的那个数据库中，已经有了一个表books，表中已经有了一条记录。

2.查询

存进去了，总要看看：

```
>>> conn = sqlite3.connect("23301.db")
>>> cur = conn.cursor()
>>> cur.execute('select * from books')
<sqlite3.Cursor object at 0xb73edea0>
>>> print cur.fetchall()
[(u'from beginner to master', u'laoqi', u'python')]
```

3.批量插入

多增加点内容，以便于做别的操作：

```
>>> books = [("first book","first","c"), ("second book","second","
```

批量插入：

```
>>> cur.executemany('insert into books values (?,?,?)', books)
<sqlite3.Cursor object at 0xb73edea0>
>>> conn.commit()
```

用循环语句打印查询结果：

```
>>> rows = cur.execute('select * from books')
>>> for row in rows:
...     print row
...
(u'from beginner to master', u'laoqi', u'python')
(u'first book', u'first', u'c')
```

```
(u'second book', u'second', u'c')
(u'third book', u'second', u'python')
```

4.更新

正如前面所说，在cur.execute()中，你可以写SQL语句来操作数据库。

```
>>> cur.execute("update books set title='physics' where author='fi
<sqlite3.Cursor object at 0xb73edea0>
>>> conn.commit()
```

按照条件查处来看一看：

```
>>> cur.execute("select * from books where author='first'")
<sqlite3.Cursor object at 0xb73edea0>
>>> cur.fetchone()
(u'physics', u'first', u'c')
```

5.删除

在操作数据的过程中，删除是必须要掌握的。

```
>>> cur.execute("delete from books where author='second'")
<sqlite3.Cursor object at 0xb73edea0>
>>> conn.commit()

>>> cur.execute("select * from books")
<sqlite3.Cursor object at 0xb73edea0>
>>> cur.fetchall()
[(u'from beginner to master', u'laoqi', u'python'), (u'physics', u
```

在你完成对数据库的操作时，一定要关门才能走人：

```
>>> cur.close()
>>> conn.close()
```

基本知识已经介绍差不多了。当然，在编程实践中，或许还会遇到问题，就请读者多参考官方文档。

7.6　电子表格

一提到电子表格，可能立刻想到的是Excel。殊不知，电子表格“历史悠久”，比Word要长久多了。根据维基百科的记载整理一个简史：

VisiCalc是第一个电子表格程序，用于苹果二号计算机。由丹·布李克林（Dan Bricklin）和鲍伯·法兰克斯顿（Bob Frankston）发展而成，1979年10月跟着苹果二号计算机推出，成为苹果二号计算机上的“杀手应用软件”。

接下来是Lotus 1-2-3，由Lotus Software（美国莲花软件公司）于1983年起所推出的电子表格软件，在DOS时期广为个人计算机用户所使用，是一套杀手级应用软件。也是世界上第一个销售超过100万套的软件。

然后微软也开始做电子表格，早在1982年，推出了其第一款电子制表软件——Multiplan，并在CP/M系统上大获成功，但在MS-DOS系统上，Multiplan败给了Lotus 1-2-3。

1985年，微软推出第一款Excel，但它只用于Mac系统；直到1987年11月，微软的第一款适用于Windows系统的Excel才诞生，不过，它一出来，就与Windows系统直接捆绑，由于此后Windows大行其道，并且Lotus1-2-3迟迟不能适用于Windows系统，到了1988年，Excel的销量超过了lotus 1-2-3。

此后就是微软的天下了，Excel后来又并入了Office里面，成为了Microsoft Office Excel。

尽管Excel已经发展了很多代，提供了大量的用户界面特性，但它仍然保留了第一款电子制表软件VisiCalc的特性：行、列组成单元格，数据、与数据相关的公式或者对其他单元格的绝对引用保存在单元格中。

由于微软独霸天下，Lotus 1-2-3已经淡出了人们的视线，甚至很多人误认为历史就是从微软开始的。

其实，除了微软的电子表格，在Linux系统中也有很好的电子表

格，Google也提供了不错的在线电子表格。

从历史到现在，电子表格都有很广泛的用途。所以，Python也要操作一番电子表格，因为有些数据，就是存在于电子表格中。

7.6.1 openpyl

openpyl模块是解决Microsoft Excel 2007/2010之类版本中扩展名是Excel 2010 xlsx/xlsm/xltx/xltm的文件的读写的第三方库。

1.安装

安装第三方库，当然用法力无边的pip install。

```
$ sudo pip install openpyxl
```

如果最终看到了下面的提示，恭喜你，安装成功。

```
Successfully installed openpyxl jdcal
Cleaning up...
```

2.workbook和sheet

第一步，引入模块，用下面的方式：

```
>>> from openpyxl import Workbook
```

接下来用Workbook()类里面的方法展开工作：

```
>>> wb = Workbook()
```

请回忆Excel文件，如果想不起来，就打开Excel，第一眼看到的是一个称之为工作簿（workbook）的东西，里面有几个sheet，默认是三个，当然可以随意增删。默认使用第一个sheet。

```
>>> ws = wb.active
```

在每个工作簿中至少有一个sheet，通过这条指令，就在当前工作簿中建立了一个sheet，并且它是当前正在使用的。

还可以在这个sheet后面追加：

```
>>> ws1 = wb.create_sheet()
```

甚至，还可以插队：

```
>>> ws2 = wb.create_sheet(1)
```

在第二个位置插入了一个sheet。

在Excel文件中一样，创建了sheet之后，默认都是以“Sheet1”、“Sheet2”的样子来命名的，然后我们可以给其重新命名。在这里，依然可以这么做。

```
>>> ws.title = "python"
```

ws所引用的sheet对象名字就是“python”了。

此时，可以使用下面的方式从工作簿对象中得到sheet

```
>>> ws01 = wb['python']          #sheet和工作簿的关系，类似键/值对的关
>>> ws is ws01
True
```

或者用这种方式：

```
>>> ws02 = wb.get_sheet_by_name("python")
>>> ws is ws02
True
```

整理一下到目前为止我们已经完成的工作：建立了工作簿（wb）和三个sheet。还是显示一下比较好：

```
>>> print wb.get_sheet_names()
['python', 'Sheet2', 'Sheet1']
```

Sheet2之所以排在了第二位，是因为在建立的时候，用了一个插队的方法。这跟在Excel中差不多，如果Sheet命名了，就按照那个名

字显示，否则就默认为名字是"Sheet1"（注意，第一个字母大写）。

也可以用循环语句，把所有的sheet名字打印出来。

```
>>> for sh in wb:
...     print sh.title
...
python
Sheet2
Sheet1
```

如果读者去dir（wb）工作簿对象的属性和方法，会发现它具有迭代的特征_iter_方法，这说明工作簿是可迭代的。

3.cell

为了能够清楚地理解填数据的过程，将电子表中约定的名称以图7-1的方式说明：

对于Sheet，其中的cell是它的下级单位。所以，要得到某个cell可以这样：

```
b4 = ws['B4']
```

如果B4这个cell已经有了，用这种方法就是将它的值赋给了变量b4；如果sheet中没有这个cell，那么就创建这个cell对象。

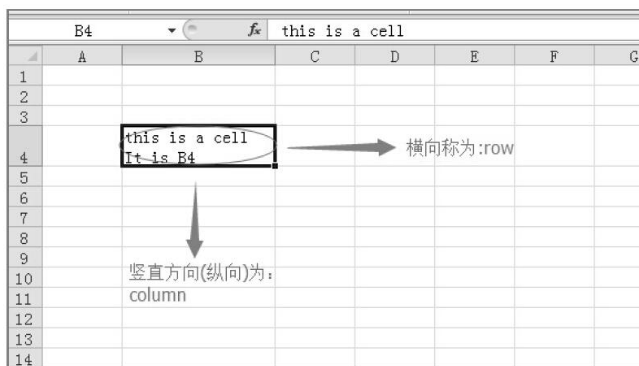


图7-1　电子表中约定的名称

请读者注意，当我们打开Excel，默认已经画好了很多cell。但

是，在Python操作的电子表格中，不会默认画好那样一个表格，一切都要创建之后才有。所以，如果按照前面的操作流程，上面就是创建了B4这个cell，并且把它作为一个对象被b4变量引用。

如果要给B4添加数据，可以这么做：

```
>>> ws['B4'] = 4444
```

因为b4引用了一个cell对象，所以可以利用这个对象的属性来查看其值：

```
>>> b4.value
4444
```

要获得（或者建立并获得）某个cell对象，还可以使用下面的方法：

```
>>> a1 = ws.cell("A1")
```

或者：

```
>>> a2 = ws.cell(row = 2, column = 1)
```

刚才已经提到，在建立了Sheet之后，内存中的它并没有cell，需要程序去建立。上面都是一个一个地建立，能不能一次建立多个呢？比如类似下面：

```
|A1|B1|C1|  |A2|B2|C2|  |A3|B3|C3|
```

就可以如同切片那样来操作：

```
>>> cells = ws["A1":"C3"]
```

可以用下面的方法查看创建结果：

```
>>> tuple(ws.iter_rows("A1:C3"))
((<Cell python.A1>, <Cell python.B1>, <Cell python.C1>),
 (<Cell python.A2>, <Cell python.B2>, <Cell python.C2>),
 (<Cell python.A3>, <Cell python.B3>, <Cell python.C3>))
```

这是按照横向顺序读过来的，即A1-B1-C1，作为一个元组，然后读下一行，再组成一个元组。还可以用下面的循环方法，一个一个地读到每个cell对象：

```
>>> for row in ws.iter_rows("A1:C3"):
...     for cell in row:
...         print cell
...
<Cell python.A1>
<Cell python.B1>
<Cell python.C1>
<Cell python.A2>
<Cell python.B2>
<Cell python.C2>
<Cell python.A3>
<Cell python.B3>
<Cell python.C3>
```

也可以用Sheet对象的rows属性，得到按照横向顺序依次排列的cell对象（注意观察结果，因为没有进行范围限制，所以目前是sheet中所有的cell，前面已经建立到第四行B4，所以，要比上面的操作多一个row）：

```
>>> ws.rows
((<Cell python.A1>, <Cell python.B1>, <Cell python.C1>),
 (<Cell python.A2>, <Cell python.B2>, <Cell python.C2>),
 (<Cell python.A3>, <Cell python.B3>, <Cell python.C3>),
 (<Cell python.A4>, <Cell python.B4>, <Cell python.C4>))
```

用sheet对象的columns属性，得到的是按照纵向顺序排列的cell对象（注意观察结果）：

```
>>> ws.columns
((<Cell python.A1>, <Cell python.A2>, <Cell python.A3>, <Cell pyth
 (<Cell python.B1>, <Cell python.B2>, <Cell python.B3>, <Cell pyth
 (<Cell python.C1>, <Cell python.C2>, <Cell python.C3>, <Cell pyth
```

不管用哪种方法，只要得到了cell对象，接下来就可以依次赋值了。比如要在上面的表格中，依次填写上1、2、3、……

```
>>> i = 1
>>> for cell in ws.rows:
...     cell.value = i
...     i += 1
```

```
... Traceback (most recent call last): File "", line 2, in Attribu
```

报错了，关键是没有注意观察上面的结果。元组里面是以元组为元素，再里面才是cell对象。所以，必须要“时时警醒”，常常谨慎。

```
>>> for row in ws.rows:
...     for cell in row:
...         cell.value = i
...         i += 1
...
```

如此，给每个cell添加了数据。查看一下，不过要换一个属性：

```
>>> for col in ws.columns:
...     for cell in col:
...         print cell.value
...
1
4
7
10
2
5
8
11
3
6
9
12
```

虽然看着有点不舒服，但的确达到了前面的要求。

4.保存

把辛苦工作的结果保存一下吧。

```
>>> wb.save("23401.xlsx")
```

如果有同名文件存在，会覆盖。

此时，可以用Excel打开这个文件，看看可视化的结果：

	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

5.读取已有文件

如果已经有一个.xlsx文件，要读取它，可以这样做：

```
>>> from openpyxl import load_workbook
>>> wb2 = load_workbook("23401.xlsx")
>>> print wb2.get_sheet_names()
['python', 'Sheet2', 'Sheet1']
>>> ws_wb2 = wb2["python"]
>>> for row in ws_wb2.rows:
...     for cell in row:
...         print cell.value
...
1
2
3
4
5
6
7
8
9
10
11
12
```

很好，就是这个文件。

7.6.2　其他第三方库

针对电子表格的第三方库，除了上面这个openpyxl之外还有别的，下面列出几个仅供参考，使用方法大同小异。

- xlsxwriter：针对Excel 2010格式，如.xlsx，官方网站：

https://xlsxwriter.readthedocs.org/，这个官方文档写得图文并茂。非常好读。

下面两个是用来处理.xls格式的电子表表格。

- xlrd：网络文件，https://secure.simplistix.co.uk/svn/xlrd/trunk/xlrd/doc/xlrd.html?p=4966。
- xlwt：网络文件，http://xlwt.readthedocs.org/en/latest/。

第3季　实战

通过前面的学习，已经掌握了Python的基本内容，不少读者可能此时已经跃跃欲试，迫切地想用已经掌握的技术去做点什么。

本季就是要讲一些实战的东西。

因为本书毕竟是一本向初学者讲述Python的教程，所以在实战中的所有例子，跟真正的工程代码要求还有一定的差距，比如可能没有非常优化，或者某些语句和方法的使用还需要进一步推敲。也盼望读者能够指出不足，必改正。

其次，所谓“实战”，多少有点纸上谈兵的味道，也就是将某些东西稍微窥探，真正深奥的东西还要等读者在实际的工程中去体会。并且，也不要寄希望在这里就能替代实践工作。

第8章　用Tornado做网站

上网干什么？登录某个网站是必不可少的。网站是谁做的呢？当然是伟大的程序员做的。网站有很多种，做网站的方式方法也有很多种。本章仅介绍利用Python语言开发网站的基本方法，而且这个网站仅具有最基本的功能。或者说，这里只是一个做网站的引子，帮读者搭建一个架子，至于里面具体的内容，还需要读者在以后的开发中自己创意。

8.1 为做网站而准备

作为一个程序员一定要会做网站，因为如果被人问及此事，而说自己不会，的确羞愧难当呀。所以，要讲一讲如何做网站。

首先，为自己准备一个服务器。这个要求似乎有点儿过分，作为一个普通的、穷困潦倒的程序员，哪里有钱来购买服务器呢？没关系，不够买服务器也能做网站，可以购买云服务空间或者虚拟空间，这个在网上搜索一下，有很多。如果连购买这个的钱也没有，还可以将自己的电脑（这总该有了）作为服务服务器。我就是利用一台装有Ubuntu操作系统的个人电脑作为本书的案例演示服务器。

然后，要在这个服务器上做一些程序配置。一些必备的网络配置这里就不说了，比如我用的Ubuntu系统，默认情况都有了。另外的配置就是Python开发环境，这个应该也有了，前面已经在用了。

接下来要安装一个框架，这里采用Tornado框架。在安装这个框架之前，先了解一些相关知识。

8.1.1 开发框架

对框架的认识，由于工作习惯和工作内容的不同，会有很大差异，这里姑且截取维基百科中的一种定义，之所以要给出一个定义，无非是让读者有所了解，但是，是否知道这个定义，丝毫不影响后面的工作。

软件框架（Software framework），通常指的是为了实现某个业界标准或完成特定基本任务的软件组件规范，也指为了实现某个软件组件规范时，提供规范所要求之基础功能的软件产品。

框架的功能类似于基础设施，与具体的软件应用无关，但是提供并实现最为基础的软件架构和体系。软件开发者通常依据特定的框架实现更为复杂的商业运用和业务逻辑。这样的软件应用可以在支持同一种框架的软件系统中运行。

简而言之，框架就是制定一套规范或者规则（思想），大家（程

序员）在该规范或者规则（思想）下工作。就好比使用别人搭好的舞台，你来做表演。

我比较喜欢最后一句解释“别人搭好舞台，我来表演”。这也就是说，在做软件开发的时候，能够减少工作量。就做网站来讲，其实需要做的事情很多，但是如果有了开发框架，很多底层的事情就不需要做了。

有些高手工程师鄙视框架，认为自己编写的才是王道。在这方面不争论，框架是开发中很流行的东西，我还是固执地认为用框架来开发更划算。

8.1.2 Python框架

有人说PHP框架多，PHP的开发框架的确很多，不过，Python的Web开发框架，也足够使用了，列举几种常见的Web框架：

- Django：这是一个被广泛应用的框架。在网上搜索，会发现很多公司在招聘的时候都要求会这个。框架只是辅助，真正的程序员，应该根据需要而做。当然不同的框架有不同的特点，需要学习一段时间。
- Flask：一个用Python编写的轻量级Web应用框架。基于Werkzeug WSGI工具箱和Jinja2模板引擎。
- Web2py：是一个为Python语言提供的全功能Web应用框架，旨在敏捷快速地开发Web应用，具有快速、安全以及可移植的数据库驱动的应用，兼容Google App Engine。
- Bottle：微型Python Web框架，遵循WSGI，说其微型，是因为它只有一个文件，除Python标准库外，它不依赖于任何第三方模块。
- Tornado：全称是Tornado Web Server，从名字上看就知道它可以用作Web服务器，但同时它也是一个Python Web的开发框架。最初是在FriendFeed公司的网站上使用，FaceBook收购了之后便开源了出来。
- webpy：轻量级的Python Web框架。webpy的设计理念力求精简（Keep it simple and powerful），源码很简短，只提供一个框架所必需的东西，不依赖大量的第三方模块，它没有URL路由、没有模板也没有数据库的访问。

以上信息选自：http://blog.jobbole.com/72306/，在这篇文章

中还有别的框架，由于不是Web框架，所以没有选摘，有兴趣的读者可以去阅读。

8.1.3　Tornado

本教程中将选择使用Tornado框架。

Tornado全称Tornado Web Server，是一个用Python语言写成的Web服务器兼Web应用框架，由FriendFeed公司在自己的网站FriendFeed中使用，被Facebook收购以后框架以开源软件的形式开放给大众。

一般用哪个框架要结合项目而定。我选用Tornado的原因，就是看中了它在性能方面的优异表现。

Tornado的性能是相当优异的，因为它试图解决一个被称之为“C10k”问题，就是处理大于或等于一万的并发。

如表8-1所示是和一些其他Web框架与服务器的对比，供读者参考（数据来源：https://developers.facebook.com/blog/post/301）。

条件：处理器为AMD Opteron，主频2.4GHz，4核。

表8-1　其他Web框架与服务器的对比

服　务	部　署	请求/每秒
Tornado	nginx，4 进程	8213
Tornado	1 个单线程进程	3353
Django	Apache/mod_wsgi	2223
web.py	Apache/mod_wsgi	2066
CherryPy	独立	785

看了这个对比表格，还有什么理由不选择Tornado呢？

影响一个网站性能的因素，不完全在于框架，还有别的因素，上面的比较仅供参考。

8.1.4　安装Tornado

Tornado的官方网站：http://www.tornadoweb.org。

我在自己的电脑中（是我目前使用的服务器），用下面的方法安装，只需要一句话即可：

```
pip install tornado
```

这是因为Tornado已经列入PyPI，因此可以通过pip或者easy_install来安装。

如果不用这种方式安装，下面的链接中有可以供读者下载的最新源码版本和安装方式：https://pypi.python.org/pypi/tornado/。

此外，在github上也有托管，读者可以通过上述页面进入到github看源码。

我没有在Windows操作系统上安装过这个，不过，在官方网站上有一句话，在告诉读者一些信息：

```
Tornado will also run on Windows, although this configuration is r
```

特别建议，在真正的工程中，网站的服务器还是用Linux比较好。

最后说明一下，要做网站，除了做好上述准备之外，还要有点别的技术准备：

- HTML
- CSS
- JavaScript

8.2　分析Hello

打开你写Python代码用的编辑器，把下面的代码一个字不差地录入进去，并命名保存为hello.py（目录自己任意定）。

```
#!/usr/bin/env python
#coding:utf-8

import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=in

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', welcome you to read: www.itdiffer

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

进入到保存hello.py文件的目录，执行：

```
$ python hello.py
```

用Python运行这个文件，其实就已经发布了一个网站，只不过这个网站太简单了。诚然，这是有前提的，那就是已经按照上一节的流程把Tornado安装好了。

接下来，打开浏览器，在浏览器中输入：http://localhost:8000，得到如图8-1所示的界面。

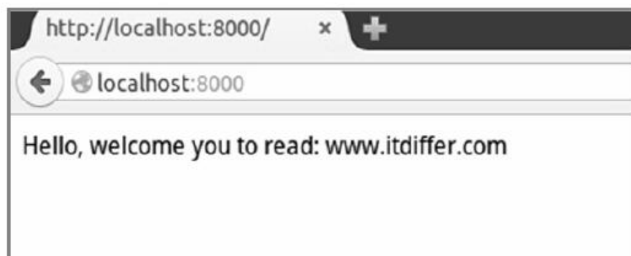


图8-1　网站界面

在Ubuntu的shell中还可以用下面的方式运行：

```
$ curl http://localhost:8000/
Hello, welcome you to read: www.itdiffer.com

$ curl http://localhost:8000/?greeting=Qiwsir
Qiwsir, welcome you to read: www.itdiffer.com
```

此操作，读者可以根据自己的系统而定。

不管怎样，都要恭喜你，迈出了决定性一步，已经可以用Tornado发布网站了。在这里似乎没有做什么部署，只是安装了Tornado。是的，不需要多做什么，因为Tornado就是一个很好的server，也是一个框架。

下面以这个非常简单的网站为例，对用Tornado做的网站的基本结构进行解释。

8.2.1　Web服务器工作流程

任何一个网站都离不开Web服务器，这里所说的不是指那个跟计算机一样的硬件设备，而是指里面安装的软件，有时候初次接触的读者容易搞混。就连伟大的维基百科都这么说：

有时，两种定义会引起混淆，如Web服务器，它可能是指用于网站的计算机，也可能是指像Apache这样的软件，运行在这样的计算机上以管理网页组件和回应网页浏览器的请求。

在具体的语境中，读者要注意分析。

在Web上，用得最多的就是输入网址，访问某个网站。全世界那么多网站网页，如果去访问，怎么能够做到彼此互通互联呢？为了协调彼此，制定了很多通用的协议，其中http协议，就是网络协议中的一种。

网上有一张图（http://kenby.iteye.com/blog/1159621），如图8-2所示，简要说明Web服务器的工作过程

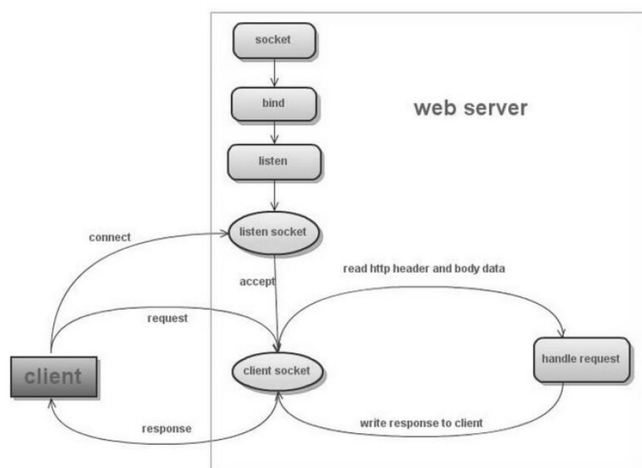


图8-2　Web服务器的工作过程

为了让读者能更理解，把原文中对图示的说明也贴上：

1.创建listen socket，在指定的监听端口，等待客户端请求的到来。

2.listen socket接受客户端的请求，得到client socket，接下来通过client socket与客户端通信。

3.处理客户端的请求，首先从client socket读取http请求的协议头，如果是post协议，还可能要读取客户端上传的数据，然后处理请求，准备好客户端需要的数据，通过client socket写给客户端。

8.2.2　解剖标本

前面跑起来的那个网站，就算是一个标本了，分析这个网站，能

让我们对网站的概况有所了解。

1.引入模块

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
```

这四个都是Tornado的模块，在本例中都是必需的。它们四个在一般的网站开发中，都要被用到，基本作用分别如下。

- tornado.httpserver：这个模块用来解决Web服务器的http协议问题，它提供了不少属性方法，实现客户端和服务器端的互通。Tornado的非阻塞、单线程的特点在这个模块中体现。
- tornado.ioloop：这个也非常重要，实现I/O循环，监听用户请求，然后映射具体的处理，再返给用户相应的结果。
- tornado.options：这是命令行解析模块，也常用到。
- tornado.web：这是必不可少的模块，它提供了一个简单的Web框架与异步功能，从而使其扩展到大量打开的连接，使其成为理想的长轮询。

读者看到这里可能有点莫名其妙，对这些东西不理解。没关系，你可以先不用管它。一定要硬着头皮一字一句地读下去，随着学习和实践的深入，现在不理解的以后会逐渐领悟。

还有一个模块引入，是用from...import完成的。

```
from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=ir
```

这两句就显示了所谓“命令行解析模块”的用途了。

通过tornado.options.define()定义了访问本服务器的端口，就是当在浏览器地址栏中输入http:localhost:8000的时候，才能访问本网站，因为http协议默认的端口是80，为了区分，在这里设置为8000，为什么要区分呢？因为我的计算机已经部署了别的（或许是Nginx、Apache）服务器了，它的端口是80，所以要区分开，并且，如果将Tornado和Nginx联合起来工作，两个服务器在同一台计算机上，就要分开（关于两者联合工作，可以到网上搜索并参考有关Tornado部署

的资料）。

2.定义请求-处理程序类

```
class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', welcome you to read: www.itdiffer
```

所谓“请求处理”程序类，就是要定义一个类，专门应付客户端向服务器提出的请求（这个请求也许是要读取某个网页，也许是要将某些信息存到服务器上），服务器要有相应的程序来接收并处理这个请求，并且反馈某些信息（或者是针对请求反馈所要的信息，或者返回其他的错误信息等）。

于是，就定义了一个类，名字是IndexHandler，名字可以随便取，但是，按照习惯类名的单词首字母都是大写的，并且如果这个类是请求处理程序类，那么最好用Handler结尾，这样在名称上很明确，望文生义，知道是干什么的。

类IndexHandler继承tornado.web.RequestHandler，其中再定义get()和post()两个在Web中应用最多的方法的内容。

在本例中，只定义了一个get()方法。

用greeting＝self.get_argument（'greeting'，'Hello'）的方式可以得到url中传递的参数，比如：

```
$ curl http://localhost:8000/?greeting=Qiwsir
Qiwsir, welcome you to read: www.itdiffer.com
```

得到了在url中为greeting设定的值Qiwsir。如果url中没有提供值，就是Hello。

官方文档对这个方法的描述如下：

RequestHandler.get_argument(name, default＝,[]strip＝True)

Returns the value of the argument with the given name.

If default is not provided, the argument is considered to be

required, and we raise a MissingArgumentError if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

接下来的那句self.write（greeting+'，weblcome you to read:www.itdiffer.com）'中，write()方法的主要功能是向客户端反馈信息。也浏览一下官方文档信息，对以后的正确理解使用有帮助：

RequestHandler.write(chunk)[source]

Writes the given chunk to the output buffer.

To write the output to the network,use the flush() method below.

If the given chunk is a dictionary,we write it as JSON and set the Content-Type of the response to be application/json. (if you want to send JSON as a different Content-Type,call set_header after calling write()).

3.main()方法

if_name_==" _main_"这个方法跟以往执行Python程序是一样的。

tornado.options.parse_command_line()，这是在执行Tornado的解析命令行。在Tornado的程序中，只要import模块之后，就会在运行的时候自动加载，不需要了解细节，但是，在main()方法中如果有命令行解析，必须提前将模块引入。

4.Application类

下面这句是重点：

```
app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
```

将tornado.web.Application类实例化。这个实例化，本质上是建

立了整个网站程序的请求处理集合，然后它可以被HTTPServer作为参数调用，实现http协议服务器访问。Application类的_init_方法参数形式：

```
def __init__(self, handlers=None, default_host="", transforms=None
    pass
```

在一般情况下，handlers是不能为空的，因为Application类要通过这个参数的值处理来自客户端的请求。例如在本例中，handlers=[（r"/"，IndexHandler）]，就意味着如果通过浏览器的地址栏输入根路径（http://localhost:8000就是根路径，如果是http://localhost:8000/qiwsir，就不属于根，而是一个子路径或目录了），对应着就是让IndexHandler类处理这个请求。

一定要注意通过handlers传入的数值格式，等到后面做复杂结构的网站时，handlers就显得重要了。它传入的是一个列表，列表里面的元素是元组，元组的组成包括两部分，一部分是请求路径，另外一部分是处理程序的类名称。注意请求路径可以用正则表达式书写（关于正则表达式，后面会进行简要介绍）。举例说明：

```
handlers = [
    (r"/", IndexHandlers),                  #来自根路径的请求用IndesHar
    (r"/qiwsir/(.*)", QiwsirHandlers),      #来自/qiwsir/以及其下任何请
                                            #QiwsirHandlers处理
]
```

在这里我使用了r"/"的样式，意味着就不需要使用转义符，r后面的都表示该符号本来的含义。例如，\n，如果单纯这么来使用，就意味着换行，因为符号"\"具有转义功能，当写成r"\n"的形式时，就不再表示换行了，而是两个字符，\和n，不会转意。一般情况下，由于正则表达式和\会有冲突，因此，当一个字符串使用了正则表达式后，最好使用此方式。

关于Application类的介绍，告一段落，还有别的参数设置没有讲，请保持耐心继续阅读后续内容。

5.HTTPServer类

实例化之后，Application对象（用app作为标签的）就可以被另外一个类HTTPServer引用，形式为：

```
http_server = tornado.httpserver.HTTPServer(app)
```

HTTPServer是tornado.httpserver里面定义的类。HTTPServer是一个单线程非阻塞HTTP服务器，执行HTTPServer一般要回调Application对象，并提供发送响应的接口，即下面的内容是跟随上面语句的（options.port的值在IndexHandler类前面通过from...import..设置）。

```
http_server.listen(options.port)
```

这种方法，就建立了单进程的http服务。

请读者牢记，如果在以后的编码中，遇到需要多进程，请参考官方文档说明：http://tornado.readthedocs.org/en/latest/httpserver.html#http-server。

6.IOLoop类

剩下最后一句了：

```
tornado.ioloop.IOLoop.instance().start()
```

这句话，总是在__main()__的最后一句。暂时不对这里的instance()和start()做深入研究（如果你去研究start()，肯定会有点突然的感觉）。

以上是一个简单的hello.py剖析。想必读者对Tornado编写网站的基本概念已经了解了。

如果还一头雾水，也不要着急，只需要有一个整体概念，不要拘泥于细节或者某些词汇含义，然后即继续学习。

8.3　做个简单的网站

从现在开始做一个网站，当然，这个网站只能算是一个毛坯的，可能很简陋，但是网站的主要元素都会涉及，读者通过此学习，能够了解网站的开发基本结构和内容，并且对前面的知识可以有综合应用。

8.3.1　基本结构

如图8-3所示是一个网站的基本结构。

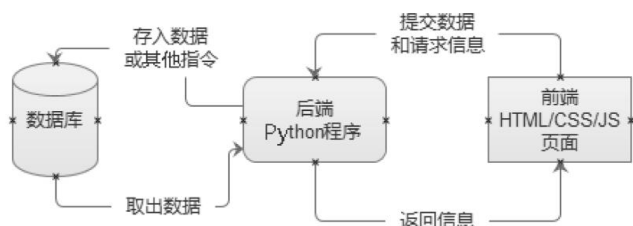


图8-3　网站的基本结构

1.前端

这是一个不很严格的说法，但是在日常开发中都这么说。在网站中，所谓前端就是指用浏览器打开之后看到的那部分，它呈现网站传过来的信息的界面，也是用户和网站之间进行信息交互的界面。撰写前端，一般使用HTML/CSS/JS，当然，非要用Python也不是不可以（例如第8.2节的例子就没有用HTML/CSS/JS），但这势必造成以后维护困难。

MVC模式是一个非常好的软件架构模式，在网站开发中，也常常要求遵守这个模式。请阅读维基百科的解释：

MVC模式（Model-View-Controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图

（View）和控制器（Controller）。

MVC模式最早由Trygve Reenskaug在1978年提出，是施乐帕罗奥多研究中心（Xerox PARC）在20世纪80年代为程序语言Smalltalk发明的一种软件设计模式。MVC模式的目的是实现一种动态的程式设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式通过对复杂度的简化，使程序结构更加直观。软件系统通过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。专业人员可以通过自身的专长分组。

- 控制器（Controller）：负责转发请求，对请求进行处理。
- 视图（View）：界面设计人员进行图形界面设计。
- 模型（Model）：程序员编写程序应有的功能（实现算法等）、数据库专家进行数据管理和数据库设计（可以实现具体的功能）。

所谓“前端”，大概对应着View部分，之所以说是大概，因为MVC是站在一个软件系统的角度进行划分的，图8-2中的前后端，与其说是系统部分的划分，不如说是系统功能的划分。

前端所实现的功能主要有：

- 呈现内容。这些内容是根据URL，由后端从数据库中提取出来的，发送给前端，然后前端将其按照一定的样式呈现出来。另外，有一些内容不是后端数据库提供的，是写在前端的。
- 用户与网站交互。比如用户登录，这是很多网站都有的功能，当用户在指定的输入框中输入信息之后，该信息就是被前端提交给后端，后端对这个信息进行处理，一般情况下都要再反馈给前端一个处理结果，然后前端呈现给用户。

2.后端

这里所说的后端对应着MVC中的Controller和Model的部分或者全部功能，因为在图8-2中，“后端”是一个狭隘的概念，没有把数据库放在其内。

不在这些术语上纠结。

后端就是用Python写的程序，主要任务是根据需要处理由前端发过来的各种请求，然后根据逻辑流程操作数据库（对数据库进行增删改查），或者把请求的处理结果反馈给前端，还可能二者兼有之。

3.数据库

工作比较单一，就是面对后端的Python程序，任其增删改查。

8.3.2　一个基本架势

我们已经制作了一个只显示一行字的网站，该网站由于功能太单一，把所有的东西都写到一个文件中。在真正的工程开发中，如果那么做，那么开发过程和后期维护会遇到麻烦，特别是不便于多人合作。所以，要做一个基本框架，以后网站就在这个框架中开发。

建立一个目录，在这个目录中建立一些子目录和文件。

```
/.
|
handlers
|
methods
|
statics
|
templates
|
application.py
|
server.py
|
url.py
```

这个结构建立好，就摆开了一个做网站的架势。有了这个架势，后面的事情就是在这个基础上添加具体内容。当然，还可以用另外一个更好听的名字——设计。

依次说明上面的架势中每个目录和文件的作用（当然，这个作用是我规定的，如果读者愿意，也可以根据自己的意愿来任意设计）。

- handlers：我准备在这个文件夹中放后端python程序，主要处理来自前端的请求，并且操作数据库。
- methods：这里准备放一些函数或者类，比如用得最多的读写数据库的函数，这些函数被handlers里面的程序使用。
- statics：这里准备放一些静态文件，比如图片、css和JavaScript文件等。
- templates：这里放模板文件，都以html为扩展名，它们将直接

面对用户。

另外，还有三个Python文件，依次写下如下内容。这些内容的功能，已经讲过，只是这里进行分门别类。

1.url.py文件

```
#!/usr/bin/env python
# coding=utf-8
"""
the url structure of website
"""

import sys
reload(sys)
sys.setdefaultencoding("utf-8")

from handlers.index import IndexHandler

url = [
    (r'/', IndexHandler),
]
```

url.py文件主要设置网站的目录结构。from handlers.index import IndexHandler，虽然在handlers文件夹还没有什么东西，为了演示如何建立网站的目录结构，假设在handlers文件夹里面已经有了一个文件index.py，它里面还有一个类IndexHandler。在url.py文件中，将其引用过来。

变量URL指向一个列表，在列表中列出所有目录和对应的处理类。比如（r'/'，IndexHandler），就是约定网站根目录的处理类是IndexHandler，即来自这个目录的get()或者post()请求，均有IndexHandler类中相应的方法来处理。

如果还有别的目录，如法炮制。

2.application.py文件

```
#!/usr/bin/env python
# coding=utf-8

from url import url
```

```
import tornado.web
import os

settings = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templ
    static_path = os.path.join(os.path.dirname(__file__), "statics
    )

application = tornado.web.Application(
    handlers = url,
    **settings
    )
```

从内容中可以看出，这个文件完成了对网站系统的基本配置，建立网站的请求处理集合。

from url import url是将url.py中设定的目录引用过来。

setting引用了一个字典对象，里面约定了模板和静态文件的路径，即声明已经建立的文件夹“templates”和“statics”分别为模板目录和静态文件目录。

接下来application就是一个请求处理集合对象。请注意tornado.web.Application()的参数设置：

```
tornado.web.Application(handlers=None, default_host='', transforms
```

关于settings的设置，不仅仅是文件中的两个参数，还可以有其他，比如，如果填上debug＝True就表示处于调试模式。调试模式的好处是开发时调试方便，但是，在正式部署的时候，最好不要用调试模式。其他更多的settings可以参看官方文档：tornado.web-RequestHandler and Application classes（http://tornado.readthedocs.org/en/latest/web.html）。

3.server.py文件

这个文件的作用是将tornado服务器运行起来，并且囊括前面两个文件中的对象属性设置。

```
#!/usr/bin/env python
# coding=utf-8

import tornado.ioloop
```

```
import tornado.options
import tornado.httpserver

from application import application

from tornado.options import define, options

define("port", default = 8000, help = "run on the given port", typ

def main():
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(options.port)

    print "Development server is running at http://127.0.0.1:%s" %
    print "Quit the server with Control-C"

    tornado.ioloop.IOLoop.instance().start()

if __name__ == "__main__":
    main()
```

如此这般，就完成了网站架势的搭建，下面要做的是向里面添加内容。

8.3.3　连接数据库

网站不一定非要有数据库，但是如果做一个功能强悍的网站，数据库就是必需的了。

接下来的网站，暂且采用MySQL数据库。

在前面已经搭建的目录结构中，找到methods，并建立一个文件db.py，然后分别建立起连接对象和游标对象。代码如下：

```
#!/usr/bin/env python
# coding=utf-8

import MySQLdb

conn = MySQLdb.connect(host="localhost", user="root", passwd="1231
cur = conn.cursor()
```

8.3.4 登录界面

很多网站上都看到用户登录功能，这里做一个简单的登录，其功能描述为：

当用户输入网址，呈现在眼前的是一个登录界面。在用户名和密码两个输入框中分别输入正确的用户名和密码之后，单击确定按钮，登录网站，显示对该用户的欢迎信息。

用图示来说明，如图8-4所示。

图8-4　用户登录界面

用户单击“登录”按钮，经过验证是合法用户之后，就呈现如图8-5所示的界面。

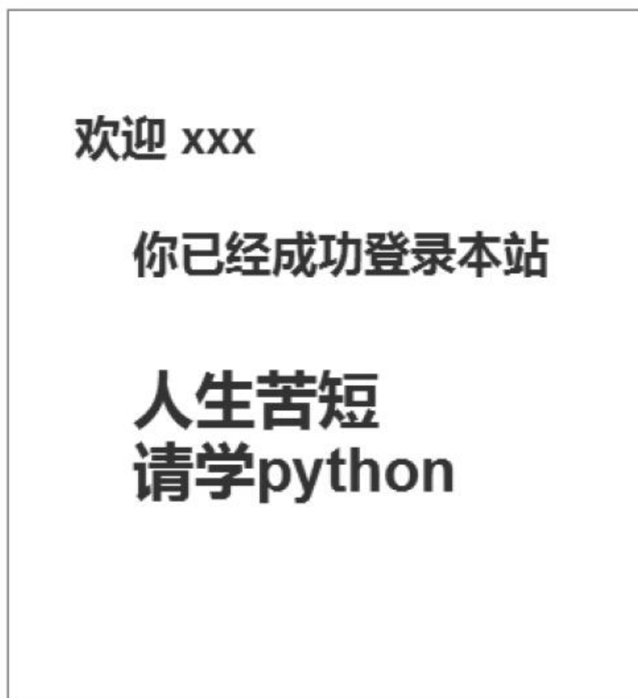


图8-5　呈现界面

先用HTML写好第一个界面。进入到templates文件，建立名为index.html的文件：

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sca
    <title>Learning Python</title>
</head>
<body>
    <h2>Login</h2>
    <form method="POST">
        <p><span>UserName:</span><input type="text" id="username"/
        <p><span>Password:</span><input type="password" id="passwo
        <p><input type="BUTTON" value="LOGIN" id="login" /></p>
    </form>
</body>
```

这是一个很简单的前端界面。要特别关注<meta name="viewport"content="width=device-width，initial-

scale＝1"/＞，其目的在于将网页的默认宽度（viewport）设置为设备的屏幕宽度（width＝device-width），并且原始缩放比例为1.0（initial-scale＝1），即网页初始大小占屏幕面积的100%。这样做的目的是让其在电脑、手机等不同大小的屏幕上，都能很好地显示。

这种样式的网页是“自适应页面”。当然，自适应页面绝非是仅仅有这样一行代码就完全解决的。要设计自适应页面，就是要进行“响应式设计”，还需要对CSS、JS乃至于其他元素如表格、图片等进行设计，或者使用一些响应式设计的框架。

一提到能够在手机上显示，读者是否想到了HTML5呢，这个被一些人热捧、被另一些人蔑视的家伙，毋庸置疑，现在已经得到了越来越广泛的应用。

HTML5是HTML最新的修订版本，2014年10月由万维网联盟（W3C）完成标准制定。目标是取代1999年所制定的HTML 4.01和XHTML 1.0标准，以期能在互联网应用迅速发展的时候，使网络标准达到符合当代的网络需求。广义论及HTML5时，实际指的是包括HTML、CSS和JavaScript在内的一套技术组合。

响应式网页设计（英语：Responsive Web Design，通常缩写为RWD），又被称为自适应网页设计、回应式网页设计。是一种网页设计的技术做法，该设计可使网站在多种浏览设备（从桌面电脑显示器到移动电话或其他移动产品设备）上阅读和导航，同时减少缩放、平移和滚动。

如果要看效果，可以直接用浏览器打开网页，因为它是.html格式的文件。

虽然完成了视觉上的设计，但是，如果单击login按钮，没有任何反应。因为它还仅仅是一个孤立的页面，这时候需要一个前端交互利器——JavaScript。

对于JavaScript，不少人对它有误解，总认为它是从Java演化出来的。它们两个有相像的地方，但其关系就如同“雷峰塔”和“雷锋”一样。详细读一读来自维基百科的诠释。

JavaScript，一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类。它的解释器被称为JavaScript引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在HTML网

页上使用，用来给HTML网页增加动态功能，然而现在也可以被用于网络服务器，如Node.js。

在1995年时，由网景公司的布兰登·艾克，在网景导航者浏览器上首次设计实现而成。因为网景公司与昇阳公司合作，网景公司管理层希望它外观看起来像Java，因此取名为JavaScript。但实际上它的语义与Self及Scheme较为接近。

为了获取技术优势，微软推出了JScript，与JavaScript同样可在浏览器上运行。为了统一规格，1997年，在ECMA（欧洲计算机制造商协会）的协调下，由网景、昇阳、微软和Borland公司组成的工作组确定统一标准：ECMA-262。因为JavaScript兼容于ECMA标准，因此也称为ECMAScript。

但是，我更喜欢用jQuery，因为它的确让我省了不少事。

jQuery是一套跨浏览器的JavaScript库，可以简化HTML与JavaScript之间的操作。由约翰·雷西格（John Resig）于2006年1月在BarCamp NYC上发布第一个版本。目前是由Dave Methvin领导的开发团队进行开发。在全球前10，000个访问最高的网站中，有65%使用了jQuery，是目前最受欢迎的JavaScript库。

在index.html文件中引入jQuery的方法有多种。

原则上，可以在HTML文件的任何地方引入jQuery库，但是通常放置的地方在html文件的开头<head>...</head>中，或者在文件的末尾</body>以内。若放在开头，如果所用的库比较大、比较多，在载入页面时的时间相对较长。

第一种引入方法是国际化的一种：

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jq
```

这是直接从jQuery CDN（Content Delivery Network）上直接引用，好处在于如果这个库更新，你不用做任何操作，就直接使用最新的了。

当然，jQuery CDN不止一个，比如官方网站的：<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>。

第二种引入方法是将jQuery下载下来，放在指定地方（比如，与

自己的网站在同一个存储器中，或者自己可以访问的另外服务器）。到官方网站（https://jqueryui.com/）下载最新的库，然后将它放在已经建立的statics目录内，为了更清楚地区分，可以在里面建立一个子目录js，jQuery库放在js子目录里面。下载的时候，建议下载以min.js结尾的文件，因为这个是经过压缩之后的，体积小。

我在statics/js目录中放置了下载的库，并且为了简短，更名为jquery.min.js。

可以用下面的方法引入：

```
<script src="statics/js/jquery.min.js"></script>
```

如果这样写也是可以的，但是考虑到Tornado的特点，用下面的方法引入更具有灵活性：

```
<script src="{{static_url("js/jquery.min.js")}}"></script>
```

不仅要引入jQuery，还需要引入自己写的js指令，所以要建立一个文件，我命名为script.js，也同时引用过来，虽然目前这个文件还是空的。

```
<script src="{{static_url("js/script.js")}}"></script>
```

这里用的static_url()是Tornado模板提供的一个函数，用这个函数，能够制定静态文件。之所以用它，而不是用上面的那种直接调用的方法，主要原因是如果某一天，将静态文件目录statics修改了，即不指定statics为静态文件目录了，定义别的目录为静态文件目录。只需要在定义静态文件目录那里修改，而其他地方的代码不需要修改。

先写一个测试性质的东西。

用编辑器打开statics/js/script.js文件，如果没有就新建。输入的代码如下：

```
$(document).ready(function(){
    alert("good");
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        alert("username: "+user);
```

```
    });
});
```

由于本书不是专门讲授JavaScript或者jQuery，所以，在js代码部分，只能一带而过，不详细解释。

上面的代码主要实现获取表单中的id值分别为username和password输入的值，alert函数的功能是把值以弹出菜单的方式显示出来。

是否还记得url.py文件？做这样的设置：

```
from handlers.index import IndexHandler

url = [
    (r'/', IndexHandler),
]
```

现在把假设有了的那个文件index.py建立起来，即在handlers里面建立index.py文件，并写入如下代码：

```
#!/usr/bin/env python
# coding=utf-8

import tornado.web

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render("index.html")
```

当访问根目录的时候，就将相应的请求交给了handlers目录中index.py文件的IndexHandler类的get()方法来处理，它的处理结果呈现index.html模板内容。

render()函数的功能在于向请求者反馈网页模板，并且可以向模板中传递数值。

将上面的文件保存之后，回到handlers目录中。因为这里面的文件要在别处被当作模块引用，所以，需要在这里建立一个空文件，命名为__init__.py。这个文件非常重要。只要在目录中加入了这个文件，该目录中的其他.py文件就可以作为模块被Python引入了。

至此，一个带有表单的网站就建立起来了。读者可以回到上一级

目录中，找到server.py文件，并运行它：

```
$ python server.py
Development server is running at http://127.0.0.1:8000
Quit the server with Control-C
```

如果读者在前面的学习中，跟我的操作完全一致，就会在shell中看到上面的结果。

打开浏览器，输入http://localhost:8000或者http://127.0.0.1:8000，看到的应该如图8-6所示。

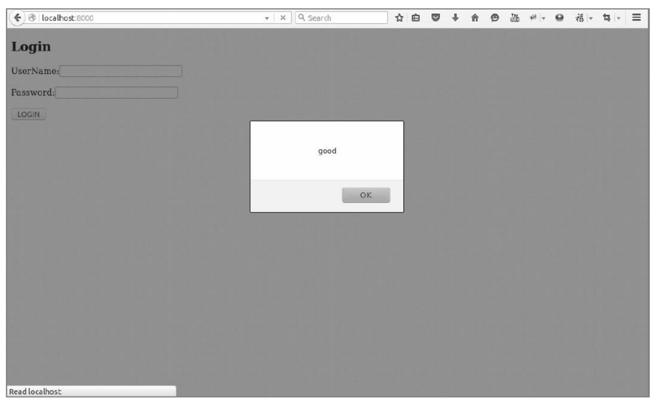


图8-6　弹出对话框

这就是script.js中的“alert（"good"）;”开始起作用了，第一句是要弹出一个对话框。单击“确定”按钮之后如图8-7所示。

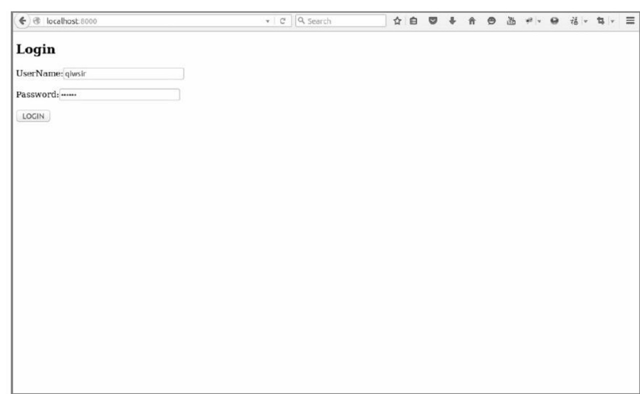


图8-7　单击“确定”按钮后的对话框

在这个页面输入用户名和密码，然后单击Login按钮，如图8-8所示。

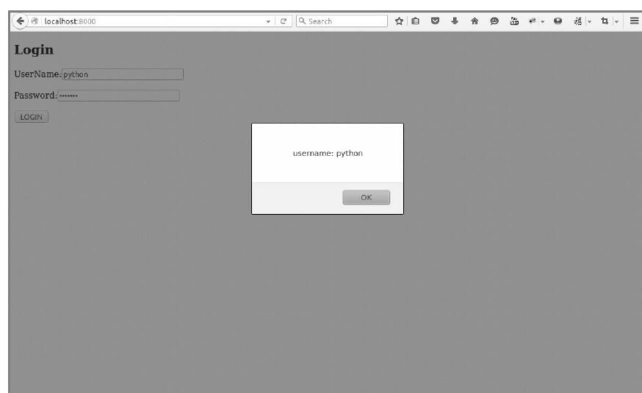


图8-8　网站雏形

一个网站有了雏形。不过，提交表单的反应，还仅仅停留在客户端，且没有向后端传递客户端的数据信息，接下来就解决这个问题。

8.3.5　数据传输

在建立了前端表单之后，就要实现前端和后端之间的数据传递。在工程中，常用到一个被称之为Ajax()的方法。

关于Ajax的故事，需要浓墨重彩，因为它足够精彩。

Ajax是“Asynchronous Javascript and XML”（异步JavaScript和XML）的缩写，在它的发展历程中，汇集了众家贡献。比如微软的IE团队曾经将XHR（XML HttpRequest）用于Web浏览器和Web服务器间传输数据，并且被W3C标准采用。当然，也有其他公司为Ajax技术做出了贡献，虽然他们都被遗忘了，比如Oddpost，后来被Yahoo！收购并成为Yahoo！Mail的基础。但是，真正让Ajax大放异彩的Google是不能被忽视的，正是Google在Gmail、Suggest和Maps上大规模使用了Ajax，才使得人们看到了它的魅力，程序员由此而兴奋。

技术总是在不断进化的，进化的方向就是用着越来越方便。

回到jQuery，里面就有Ajax()方法，能够让程序员方便地调用。

Ajax()方法通过HTTP请求加载远程数据。

该方法是jQuery底层AJAX实现。简单易用的高层实现如$.get、$.post等。$.ajax()返回其创建的XMLHttpRequest对象。大多数情况下你无须直接操作该函数，除非你需要操作不常用的选项，以获得更多的灵活性。

最简单的情况下，$.ajax()可以不带任何参数直接使用。

在上文介绍Ajax的时候，用到了一个重要的术语——“异步”，与之相对应的叫作“同步”，对此引用来自阮一峰的网络日志中的通俗描述：

“同步模式”就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；“异步模式”则完全不同，每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。

“异步模式”非常重要。在浏览器端，耗时很长的操作都应该异步执行，避免浏览器失去响应，最好的例子就是Ajax操作。在服务器端，“异步模式”甚至是唯一的模式，因为执行环境是单线程的，如果允许同步执行所有http请求，服务器性能会急剧下降，很快就会失去响应。

看来，Ajax()是前后端进行数据传输的重要角色。

承接前面对简单网站的研发，接下来是用Ajax()方法实现前后端的数据传输，只需要修改script.js文件内容即可：

```
$(document).ready(function(){
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        var pd = {"username":user, "password":pwd};
        $.ajax({
            type:"post",
            url:"/",
            data:pd,
            cache:false,
```

```
                success:function(data){
                    alert(data);
                },
                error:function(){
                    alert("error!");
                },
            });
        });
    });
```

在这段代码中，“var pd＝{"username":user，"password":pwd};”是将得到的user和pwd值，放到一个json对象中。接下来就是利用Ajax()方法将这个json对象传给后端。

jQuery中的Ajax()方法使用比较简单，正如上面的代码所示，只需要$.ajax()即可，不过需要对里面的参数进行说明。

- type：是post还是get。
- url：post或者get的地址。
- data：传输的数据，包括三种，（1）html拼接的字符串；（2）json数据；（3）form表单经serialize()序列化的。本例中传输的就是json数据，这也是经常用到的一种方式。
- cache：默认为True，如果不允许缓存，设置为False。
- success：请求成功时执行回调函数。本例中，将返回的data用alert方式弹出来。读者是否注意到，我在很多地方都用了alert()这个东西，目的在于调试，走一步看一步，看看得到的数据是否是自己所要。
- error：请求失败所执行的函数。

8.3.6　数据处理

前端通过Ajax技术，将数据以json格式传给了后端，并且指明了对象目录"/"，这个目录在url.py文件中已经做了配置，是由handlers目录中index.py文件的IndexHandler类来处理。因为是用post方法传的数据，那么在这个类中就要有post方法来接收数据。所以，要在IndexHandler类中增加post()，增加之后的完善代码是：

```
#!/usr/bin/env python
# coding=utf-8
```

```
import tornado.web

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render("index.html")

    def post(self):
        username = self.get_argument("username")
        password = self.get_argument("password")
        self.write(username)
```

在post()方法中，使用get_argument()函数来接收前端传过来的数据，这个函数的完整格式是get_argument（name，default＝[]，strip＝True），它能够获取name的值。在上面的代码中，name就是从前端传到后端的那个json对象的键的名字，是哪个键就获取哪个键的值。如果获取不到name的值，就返回default的值，但是这个值默认是没有的，如果真的没有就会抛出HTTP 400。特别注意，在get的时候，通过get_argument()函数获得url的参数，如果是多个参数，就获取最后一个的值。要想获取多个值，可以使用get_arguments（name，strip＝True）。

上例中分别用get_argument()方法得到了username和password，并且它们都是unicode编码的数据。

tornado.web.RequestHandler的方法write()，即上例中的self.write（username），是后端向前端返回数据。这里返回的实际上是一个字符串，也可返回json字符串。

如果读者要查看修改代码之后的网站效果，最有效的方式是先停止网站（ctrl＋c），再重新执行python server.py运行网站，然后刷新浏览器即可。这是一种较为笨拙的方法。一种灵巧的方法是开启调试模式。在设置setting的时候，写上debug＝True就表示是调试模式了。但是，调试模式也不是十全十美，如果修改模板，就不会加载，还需要重启服务。

看看上面的代码效果，如图8-9所示。

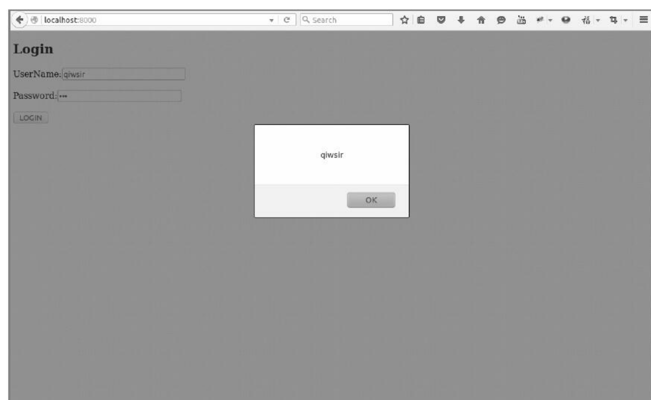


图8-9　代码效果

前端输入了用户名和密码之后，单击login按钮，提交给后端，后端再向前端返回数据之后的效果。这就是我们想要的结果。

按照流程，用户在前端输入了用户名和密码，并通过Ajax提交到了后端，后端借助于get_argument()方法得到了所提交的数据（用户名和密码）。下面要做的事情就是验证这个用户名和密码是否合法，其体现在：

- 数据库中是否有这个用户。
- 密码和用户先前设定的密码（已经保存在数据库中）是否匹配。

这个验证工作完成之后，才能允许用户登录，登录之后才能继续做某些事情。

首先，在methods目录中（已经有了一个db.py）创建一个文件，我将其命名为readdb.py，专门用来存储读数据用的函数（这种划分完全是为了明确和演示一些应用方法，读者也可以都写到db.py中）。这个文件的代码如下：

```
#!/usr/bin/env python
# coding=utf-8

from db import *

def select_table(table, column, condition, value ):
    sql = "select " + column + " from " + table + " where " + cond
    cur.execute(sql)
```

```
    lines = cur.fetchall()
    return lines
```

上面这段代码，建议读者写上注释，以检验自己是否能够将以往的知识融会贯通地应用。

有了这段代码之后，就进一步改写index.py中的post()方法。为了明了，将index.py的全部代码呈现如下：

```
#!/usr/bin/env python
# coding=utf-8

import tornado.web
import methods.readdb as mrd

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render("index.html")

    def post(self):
        username = self.get_argument("username")
        password = self.get_argument("password")
        user_infos = mrd.select_table(table="users",column="*",cor
        if user_infos:
            db_pwd = user_infos[0][2]
            if db_pwd == password:
                self.write("welcome you: " + username)
            else:
                self.write("your password was not right.")
        else:
            self.write("There is no thi user.")
```

特别注意，在methods目录中，只有不缺少_init_.py文件，才能在index.py中实现import methods.readdb as mrd。

代码修改到这里，看到的结果如图8-10所示。

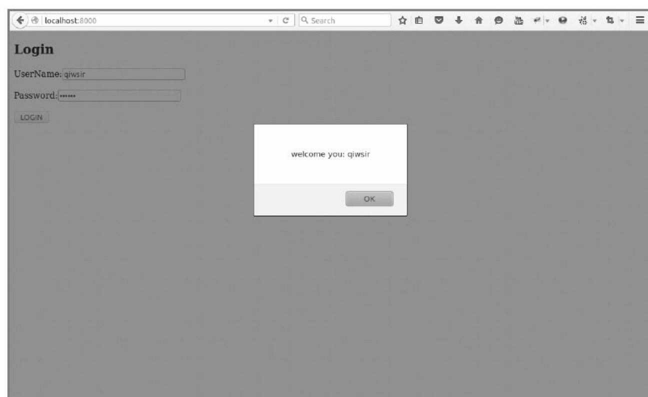


图8-10　修改代码的结果

如图8-11所示是正确输入用户名（所谓正确，就是输入的用户名和密码合法，即在数据库中有该用户名，且密码匹配），并提交数据后，反馈给前端的欢迎信息。

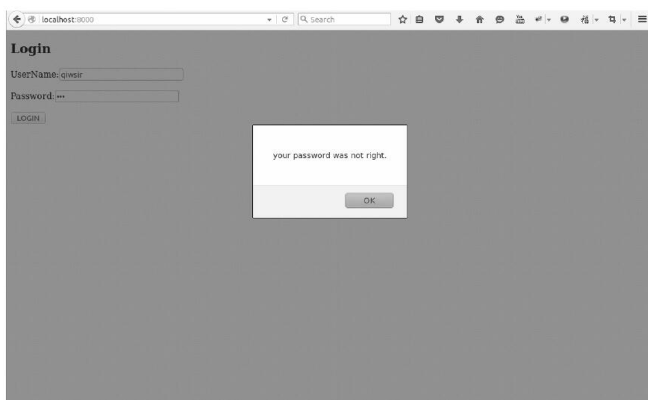


图8-11　欢迎信息

用户的输入是最不可靠的，或许会出现多种情况。

如图8-12所示是输入的密码错误了，前端反馈给用户提示的信息。

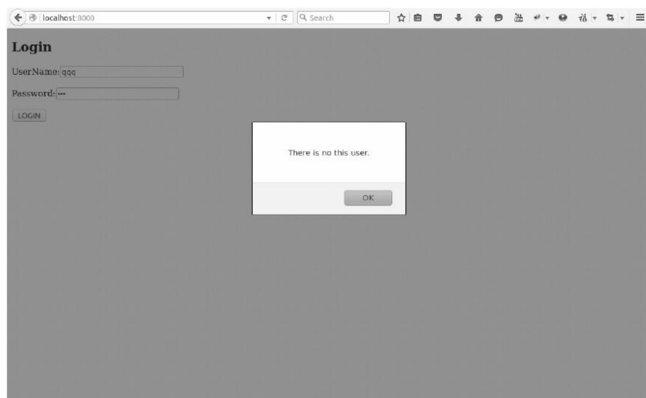


图　8-12

这是随意输入的结果，数据库中无此用户。

上述演示中，数据库中的用户密码并没有加密，这不是真实的开发行为，在真实的开发中，一定要加密传输。

8.3.7　模板

网站做到现在，突然发现，前端页面写得太难看了。俗话说“外行看热闹，内行看门道”。程序员写的网站，在更多时候是给“外行”看的，他们可没有耐心来看代码，他们看的就是界面，因此把界面做得漂亮一点点至关重要。

其实，也不仅仅是漂亮的原因，而且前端页面还要显示从后端读取出来的数据。

恰好，Tornado提供比较好用的前端模板（tornado.template），通过这个模板，能够让前端编写更方便。

- render()

render()方法能够告诉Tornado读入哪个模板，插入其中的模板代码，并返回结果给浏览器。比如在IndexHandler类中get()方法里面的self.render（"index.html"），就是让Tornado到templates目中找到名为index.html的文件，读出它的内容，返回给浏览器。这样用户就能看到index.html所规定的页面了。前面所写的index.html仅仅是html标记，没有显示出所谓“模板”的作用。为此，将index.html和

index.py文件做如下改造。

```
#!/usr/bin/env python
# coding=utf-8

import tornado.web
import methods.readdb as mrd

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        usernames = mrd.select_columns(table="users",column="usern
        one_user = usernames[0][0]
        self.render("index.html", user=one_user)
```

index.py文件中，只修改了get()方法，从数据库中读取用户名，并且提出用户（one_user），然后通过self.render（"index.html"，user＝one_user）将这个用户名放到index.html中，其中user＝one_user的作用就是传递对象到模板。

要提醒读者注意的是，在上面的代码中，我使用了mrd.select_columns（table＝"users"，column＝"username"），也就是说必须要在methods目录中的readdb.py文件中有一个名为select_columns的函数。为了使读者能够理解，贴出已经修改的readdb.py文件代码，比上一节多了函数select_columns：

```
#!/usr/bin/env python
# coding=utf-8

from db import *

def select_table(table, column, condition, value ):
    sql = "select " + column + " from " + table + " where " + cond
    cur.execute(sql)
    lines = cur.fetchall()
    return lines

def select_columns(table, column ):
    sql = "select " + column + " from " + table
    cur.execute(sql)
    lines = cur.fetchall()
    return lines
```

下面是index.html修改后的代码：

```
<!DOCTYPE html>
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sca
    <title>Learning Python</title>
</head>
<body>
    <h2>登录页面</h2>
    <p>用用户名为：{{user}}登录</p>
    <form method="POST">
        <p><span>UserName:</span><input type="text" id="username"/
        <p><span>Password:</span><input type="password" id="passwo
        <p><input type="BUTTON" value="登录" id="login" /></p>
    </form>
    <script src="{{static_url("js/jquery.min.js")}}"></script>
    <script src="{{static_url("js/script.js")}}"></script>
</body>
```

“<p>用户名为：{{user}}登录</p>”，在这里用了“{{}}”方式接受对应的变量引用的对象。即在首页打开之后，用户应当看到有一行提示，如图8-13所示。



图8-13　首页提示

图中箭头所指就是从数据库中读取出来的用户名，借助于模板中的双大括号“{{}}”显示出来。

“{{}}”本质上是占位符，当这个html被执行的时候，这个位置会被一个具体的对象（例如上面就是字符串qiwsir）所替代。具体是哪

个具体对象替代这个占位符，完全由render()方法中的关键词来指定，也就是render()中的关键词与模板中的占位符包裹着的关键词一致。

用这种方式，修改一下用户正确登录之后的效果。要求用户正确登录之后，跳转到另外一个页面，并且在那个页面中显示出用户的完整信息。

先修改url.py文件，在其中增加一些内容。完整代码如下：

```
#!/usr/bin/env python
# coding=utf-8
"""
    the url structure of website
"""
import sys
reload(sys)
sys.setdefaultencoding("utf-8")

from handlers.index import IndexHandler
from handlers.user import UserHandler

url = [
    (r'/', IndexHandler),
    (r'/user', UserHandler),
]
```

然后就建立handlers/user.py文件，内容如下：

```
#!/usr/bin/env python
# coding=utf-8

import tornado.web
import methods.readdb as mrd

class UserHandler(tornado.web.RequestHandler):
    def get(self):
        username = self.get_argument("user")
        user_infos = mrd.select_table(table="users",column="*",cor
        self.render("user.html", users = user_infos)
```

在get()中使用self.get_argument（"user"），目的是要通过url获取参数user的值。因此，当用户登录后，得到正确的返回值，那么js应该用这样的方式载入新的页面。

注意，上述的user.py代码为了简单仅突出本将要说明的，没有对user_infos的结果进行判断，但在实际的编程中，需要进行判断或者使用try…except。

```
$(document).ready(function(){
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        var pd = {"username":user, "password":pwd};
        $.ajax({
            type:"post",
            url:"/",
            data:pd,
            cache:false,
            success:function(data){
                window.location.href = "/user?user="+data;
            },
            error:function(){
                alert("error!");
            },
        });
    });
});
```

接下来是user.html模板。注意，上面的代码中，user_infos引用的对象不是一个字符串了，即传入模板的不是一个字符串，而是一个元组。对此，模板这样来处理它：

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sca
    <title>Learning Python</title>
</head>
<body>
    <h2>Your informations are:</h2>
    <ul>
        {% for one in users %}
            <li>username:{{one[1]}}</li>
            <li>password:{{one[2]}}</li>
            <li>email:{{one[3]}}</li>
        {% end %}
    </ul>
</body>
```

显示的效果如图8-14所示。



图8-14　显示的效果

在上面的模板中，其实用到了模板语法。

在模板的双大括号中，可以写类似Python的语句或者表达式。比如：

```
>>> from tornado.template import Template
>>> print Template("{{ 3+4 }}").generate()
7
>>> print Template("{{ 'python'[0:2] }}").generate()
py
>>> print Template("{{ '-'.join(str(i) for i in range(10)) }}").ge
0-1-2-3-4-5-6-7-8-9
```

如果在模板中的某个地方写上{{3＋4}}，当那个模板被render()读入之后，在页面上该占位符的地方就显示7。这说明Tornado自动将双大括号内的表达式进行计算，并将其结果以字符串的形式返回到浏览器输出。

除了表达式之外，Python的语句也可以在表达式中使用，包括if、for、while和try。只不过要有一个语句做开始和结束的标记，用以区分哪里是语句、哪里是HTML标记符。

语句的形式：{{%语句%}}

例如：

```
{{% if user=='qiwsir' %}}
    {{ user }}
{{% end %}}
```

上面的举例中，第一行虽然是if语句，但是不要在后面写冒号了。最后一行一定不能缺少，表示语句块结束。将这一个语句块放到模板中，当被render读取此模板的时候，Tornado将执行结果返回给浏览器显示，跟前面的表达式一样。实际例子中可以看上图的输出结果和对应的循环语句。

8.3.8 转义字符

虽然读者已经对字符转义问题不陌生了，但是在网站开发中，它还是一个令人感到麻烦的问题。转义字符（Escape Sequence）也称为字符实体（Character Entity），它的存在是因为在网页中“＜”、“＞”之类的符号不能直接被输出，因为它们已经被用作了HTML标记符，如果在网页上用到它们，就要转义。另外，还有一些字符在ASCII字符集中没有定义（如版权符号“©”），若这样的符号在HTML中出现，也需要转义字符（如“©”对应的转义字符是“©”）。

上述是指前端页面的字符转义，在后端程序中，因为要读写数据库，也会遇到字符转义问题。

比如一个简单的查询语句“select username，password from usertable where username＝'qiwsir'”，如果在登录框中没有输入“qiwsir”，而是输入了“a;drop database;”，这个查询语句就变成了“select username，password from usertable where username＝a;drop database;”，如果后端程序执行了这条语句会怎么样呢？后果很严重，因为会drop database，届时真的是欲哭无泪了。类似的情况还很多，比如还可以输入“＜input type＝"text"/＞”，结果出现了一个输入框，如果是“＜form action＝"..."”，就会造成跨站攻击。这方面的问题还很多，读者有空可以到网上搜索一下。

所以，后端也要转义。转义是不是很麻烦呢？

Tornado为你着想了，因为存在以上转义问题，而且会有粗心的程序员忘记，于是在Tornado中，模板默认为自动转义，这是多么好的设计呀。于是所有表单输入，你就不用担心会遇到上述问题了。

为了能够体会自动转义，不妨在登录框中输入上面那样的字符，然后用print语句看看后台得到了什么（请读者自行完成）。

自动转义是一个好事情，但是，有时候不需要转义，比如想在模板中这样做：

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sca
    <title>Learning Python</title>
</head>
<body>
    <h2>登录页面</h2>
    <p>用用户名为：{{user}}登录</p>
    <form method="POST">
        <p><span>UserName:</span><input type="text" id="username"/
        <p><span>Password:</span><input type="password" id="passwo
        <p><input type="BUTTON" value="登录" id="login" /></p>
    </form>
    {% set website = "<a href='http://www.itdiffer.com'>welcome to
    {{ website }}
    <script src="{{static_url("js/jquery.min.js")}}"></script>
    <script src="{{static_url("js/script.js")}}"></script>
</body>
```

这是index.html的代码，我增加了{%set website＝"＜a href＝'http://www.itdiffer.com'＞welcome to my website＜/a＞"%}，作用是设置一个变量，名字是website，它对应的内容是一个做了超链接的文字。然后在下面使用这个变量{{website}}，本希望能够出现的是一行字“welcome to my website”，单击这行字，就可以打开对应链接的网站。可是，看到了如图8-15所示的页面。



图8-15　自动转义的结果

下面那一行把整个源码都显示出来了，这就是自动转义的结果。这里需要的是不转义。于是可以将{{website}}修改为：

```
{% raw website %}
```

表示这一行不转义。但是别的地方还是转义的。这是一种最推荐的方法。

如果你要全转义，可以使用：

```
{% autoescape None %}
{{ website }}
```

貌似省事，但是并不推荐使用。

将下面几个函数放在这里备查，或许在某些时候会用到，都是可以使用在模板中的。

- escape（s）：替换字符串s中的&、＜、＞为他们对应的HTML字符。
- url_escape（s）：使用urllib.quote_plus替换字符串s中的字符为url编码形式。
- json_encode（val）：将val编码成JSON格式。
- squeeze（s）：过滤字符串s，把连续的多个空白字符替换成一个空格。

8.3.9　模板继承

用前面的方法已经能够很顺利地编写模板了。如果读者留心一下，会觉得每个模板都有相同的内容，遇到这种问题，作为程序员应该想到“继承”，它的作用之一就是能够让代码重用。

在Tornado的模板中，也能继承。

先建立一个文件，命名为base.html，代码如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sca
    <title>Learning Python</title>
</head>
<body>
    <header>
        {% block header %}{% end %}
    </header>
    <content>
        {% block body %}{% end %}
    </content>
    <footer>
        {% set website = "<a href='http://www.itdiffer.com'>welcom
        {% raw website %}
    </footer>
    <script src="{{static_url("js/jquery.min.js")}}"></script>
    <script src="{{static_url("js/script.js")}}"></script>
</body>
</html>
```

接下来就以base.html为父模板，依次改写index.html模板和user.html模板。

index.html代码如下：

```
{% extends "base.html" %}

{% block header %}
    <h2>登录页面</h2>
    <p>用用户名为：{{user}}登录</p>
{% end %}
{% block body %}
```

```
    <form method="POST">
        <p><span>UserName:</span><input type="text" id="username"/
        <p><span>Password:</span><input type="password" id="passwo
        <p><input type="BUTTON" value="登录" id="login" /></p>
    </form>
{% end %}
```

user.html的代码如下：

```
{% extends "base.html" %}

{% block header %}
    <h2>Your informations are:</h2>
{% end %}

{% block body %}
    <ul>
        {% for one in users %}
            <li>username:{{one[1]}}</li>
            <li>password:{{one[2]}}</li>
            <li>email:{{one[3]}}</li>
        {% end %}
    </ul>
{% end %}
```

以上代码已经没有以前重复的部分了。“{%extends"base.html"%}”意味着以base.html为父模板。在base.html中规定了形式如同“{%block header%}{%end%}”这样的块语句，在index.html和user.html中，分别对块语句中的内容进行了重写（或者说是填充）。这就相当于在base.html中做了一个结构，在子模板中按照这个结构填内容。

8.3.10 CSS

基本的流程已经差不多了，如果要美化前端，还需要使用css，它的使用方法跟js类似，也是在静态目录中建立文件即可。然后把下面这句加入到base.html的<head></head>中：

```
<link rel="stylesheet" type="text/css" href="{{static_url("css/sty
```

当然，要在style.css中写一个样式，比如：

```
body {
    color:red;
}
```

然后看看前端显示什么样子了，如图8-16所示。



图8-16　前端显示

至于其他关于CSS方面的内容，就不重点讲解了，读者可以参考关于CSS的资料。

至此，一个简单的基于Tornado的网站就做好了，虽然它很丑，但是它很有前途。因为读者只要按照上述的讨论，就可以在里面增加各种自己认为可以增加的内容。

建议读者在学习以上内容基础上，可以继续完成下面的几个功能：

- 用户注册。
- 用户发表文章。
- 用户文章列表，并根据文章标题查看文章内容。
- 用户重新编辑文章。

8.3.11　cookie和安全

cookie是现在网站重要的内容，特别是当有用户登录的时候，所以需要学习了解cookie。维基百科如是说：

cookie（复数形态cookies），中文名称为小型文本文件或小甜饼，指某些网站为了辨别用户身份而储存在用户本地终端（Client Side）上的数据（通常经过加密）。定义于RFC2109。是网景公司的前雇员Lou Montulli在1993年3月发明的。

关于cookie的作用，维基百科说得非常详细：

因为HTTP协议是无状态的，即服务器不知道用户上一次做了什么，这严重阻碍了交互式Web应用程序的实现。在典型的网上购物场景中，用户浏览了几个页面，买了一盒饼干和两瓶饮料。最后结账时，由于HTTP的无状态性，不通过额外的手段，服务器并不知道用户到底买了什么。所以cookie就是用来绕开HTTP的无状态性的“额外手段”之一。服务器可以设置或读取cookies中包含的信息，借此维护用户跟服务器会话中的状态。

在刚才的购物场景中，当用户选购了第一项商品，服务器在向用户发送网页的同时，还发送了一段cookie，记录着那项商品的信息。当用户访问另一个页面，浏览器会把cookie发送给服务器，于是服务器就知道他之前选购了什么。用户继续选购饮料，服务器就在原来那段cookie里追加新的商品信息。结账时，服务器读取发送来的cookie就行了。

cookie另一个典型的应用是，当登录一个网站时网站往往会请求用户输入用户名和密码，并且用户可以勾选“下次自动登录”。如果勾选了，那么下次访问同一网站时，用户会发现没输入用户名和密码就已经登录了。这正是因为前一次登录时，服务器发送了包含登录凭据（用户名加密码的某种加密形式）的cookie到用户的硬盘上。第二次登录时（如果该cookie尚未到期）浏览器会发送该cookie服务器验证凭据，于是不必输入用户名和密码就让用户登录了。

cookie也有缺陷，比如来自伟大的维基百科也列出了三条：

- cookie会被附加在每个HTTP请求中，所以无形中增加了流量。
- 由于在HTTP请求中的cookie是明文传递的，所以安全性成问题（除非用HTTPS）。
- cookie的大小限制在4KB左右，或许在某些情况下有点不够用。

对于用户来说，可以通过改变浏览器的设置来禁用cookie，也可以删除历史的cookie。但就目前而言，大多数人都不再禁用cookie了。

cookie最让人担心的还是由于它存储了用户的个人信息，并且最终这些信息要发给服务器，那么它就会成为某些人的目标或者工具，比如有cookie盗贼，就是搜集用户cookie，然后利用这些信息进入用户账号，达到个人某种不可告人之目的；还有被称之为cookie投毒的说法，是利用客户端的cookie传给服务器的机会，修改传回去的值。这些行为常常是通过一种被称为“跨站指令脚本（Cross site scripting）”（或者跨站指令码）的行为方式实现的。伟大的维基百科这样解释了跨站脚本：

跨网站脚本（Cross-site scripting，通常简称为XSS或跨站脚本或跨站脚本攻击）是一种网站应用程序的安全漏洞攻击，是代码注入的一种。它允许恶意用户将代码注入到网页上，其他用户在观看网页时就会受到影响。这类攻击通常包含HTML和用户端脚本语言。

XSS攻击通常指的是利用网页开发时留下的漏洞，通过巧妙的方法注入恶意指令代码到网页，使用户加载并执行攻击者恶意制造的网页程序。这些恶意网页程序通常是JavaScript，但实际上也可以包括Java、VBScript、ActiveX、Flash或者是普通的HTML。攻击成功后，攻击者可能得到更高的权限（如执行一些操作）、私密网页内容、会话和cookie等各种内容。

对cookie的普遍使用，用户和网站都受益了，但也要防止有人用它作恶。

在Tornado中，也提供对cookie的读写函数，帮助我们管理和使用它。

set_cookie()和get_cookie()是默认提供的两个方法，但它是明文不加密传输的。

在index.py文件的IndexHandler类的post()方法中，当用户登录，验证用户名和密码后，将用户名和密码存入cookie，代码如下：

```
def post(self):
    username = self.get_argument("username")
    password = self.get_argument("password")
    user_infos = mrd.select_table(table="users", column="*", condi
    if user_infos:
```

```
        db_pwd = user_infos[0][2]
        if db_pwd == password:
            self.set_cookie(username,db_pwd)      #设置cookie
            self.write(username)
        else:
            self.write("your password was not right.")
    else:
        self.write("There is no thi user.")
```

上面代码中，较以前只增加了一句“self.set_cookie（username，db_pwd）”，再回到登录页面，运行之后如图8-17所示。

看图8-17中箭头所指，从左开始的第一个是用户名，第二个是存储的该用户密码。将我在登录时输入的密码以明文的方式存储在cookie里面了。

明文存储，显然不安全。

Tornado提供另外一种安全的方法：set_secure_cookie()和get_secure_cookie()，之所以称其为安全cookie，是因为它以明文加密的方式传输。此外，跟set_cookie()的区别还在于，set_secure_cookie()执行后的cookie保存在磁盘中，直到它过期为止。也是因为这个原因，即使关闭浏览器，在失效时间以前，cookie都一直存在。

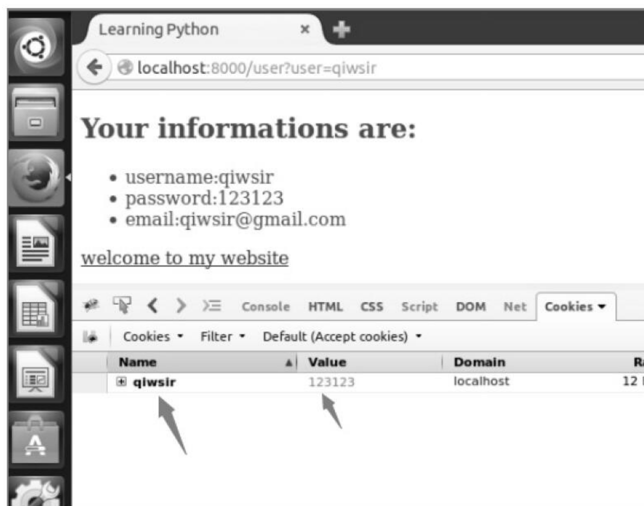


图8-17　回到登录页面

要是用set_secure_cookie()方法设置cookie，要先在application.py文件的setting中进行如下配置：

```
setting = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templ
    static_path = os.path.join(os.path.dirname(__file__), "statics
    cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E="
    )
```

其中“cookie_secret＝"bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E＝"”是为此增加的，但是，它并不是真正的加密，仅仅是一个障眼法罢了。

因为tornado会将cookie值编码为Base-64字符串，并增加一个时间戳和一个cookie内容的HMAC签名。所以，cookie_secret的值，常常用下面的方式生成（这是一个随机的字符串）：

```
>>> import base64, uuid
>>> base64.b64encode(uuid.uuid4().bytes)
'w8yZud+kRHiP9uABEXaQiA=='
```

如果嫌弃上面的签名短，可以用“base64.b64encode（uuid.uuid4().bytes＋uuid.uuid4().bytes）”获取。这里得到的是一个随机字符串，用它作为cookie_secret值。

然后修改index.py中设置cookie那句话，变成：

```
self.set_secure_cookie(username,db_pwd)
```

重新跑一个，效果如图8-18所示。

啊哈，果然“密”了很多。

如果要获取此cookie，用self.get_secure_cookie（username）即可。



图8-18　修改后效果图

这样是不是就安全了？如果这样就安全了，那你也太低估黑客们的技术实力了，甚至于用户自己也会修改cookie值。还要更安全，所以又有了httponly和secure属性，用来防范cookie投毒。设置方法是：

```
self.set_secure_cookie(username, db_pwd, httponly=True, secure=Tru
```

要获取cookie，可以使用self.set_secure_cookie（username）方法，将这句放在user.py中某个适合的位置，并且可以用print语句打印出结果，就能看到变量username对应的cookie了。这时候已经不是那个“密”过的，是明文显示。

用这样的方法，浏览器通过SSL连接传递cookie，能够在一定程度上防范跨站脚本攻击。

8.3.12　XSRF

XSRF的含义是Cross-site request forgery，即跨站请求伪造，也称之为“one click attack”，通常缩写成CSRF或者XSRF，可以读作“sea surf”。这种对网站的攻击方式跟上面的跨站脚本（XSS）似乎相像，但攻击方式不一样。XSS利用站点内的信任用户，而XSRF则通过伪装来自受信任用户的请求而利用受信任的网站。与XSS攻击相比，XSRF攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比XSS更具危险性。

还有一点需要提醒读者，即在开发应用时需要深谋远虑。任何会

产生副作用的HTTP请求，比如单击购买按钮、编辑账户设置、改变密码或删除文档等都应该使用post()方法，这是良好的RESTful做法。

又一个新名词：REST。这是一种Web服务实现方案。伟大的维基百科中这样描述：

表征性状态传输（英文：Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。目前在三种主流的Web服务实现方案中，因为REST模式与复杂的SOAP和XML-RPC相比更加简洁，越来越多的Web服务开始采用REST风格设计和实现。例如，Amazon.com提供接近REST风格的Web服务进行图书查找；雅虎提供的Web服务也是REST风格的。

在Tornado中还提供了XSRF保护的方法。

在application.py文件中使用xsrf_cookies参数开启XSRF保护。

```
setting = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templ
    static_path = os.path.join(os.path.dirname(__file__), "statics
    cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E="
    xsrf_cookies = True,
)
```

这样设置之后，Tornado将拒绝请求参数中不包含正确的_xsrf值的post/put/delete请求。Tornado会在后面悄悄地处理_xsrf cookies，所以，在表单中也要包含XSRF令牌以确保请求合法。比如index.html的表单，修改如下：

```
{% extends "base.html" %}

{% block header %}
    <h2>登录页面</h2>
    <p>用用户名为：{{user}}登录</p>
{% end %}
{% block body %}
    <form method="POST">
        {% raw xsrf_form_html() %}
        <p><span>UserName:</span><input type="text" id="username"/
        <p><span>Password:</span><input type="password" id="passwo
        <p><input type="BUTTON" value="登录" id="login" /></p>
    </form>
{% end %}
```

“{%raw xsrf_form_html()%}”是新增的，目的就在于实现上面所说的授权给前端以合法请求。

前端向后端发送的请求是通过Ajax()，所以，在Ajax请求中，需要一个_xsrf参数。

以下是script.js的代码：

```
function getCookie(name){
    var x = document.cookie.match("\\b" + name + "=([^;]*)\\b");
    return x ? x[1]:undefined;
}

$(document).ready(function(){
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        var pd = {"username":user, "password":pwd, "_xsrf":getCook
        $.ajax({
            type:"post",
            url:"/",
            data:pd,
            cache:false,
            success:function(data){
                window.location.href = "/user?user="+data;
            },
            error:function(){
                alert("error!");
            },
        });
    });
});
```

函数getCookie()的作用是得到cookie值，然后将这个值放到向后端post的数据中“var pd＝{"username":user，"password":pwd，"_xsrf":getCookie（"_xsrf"）};”。运行的结果如图8-19所示。

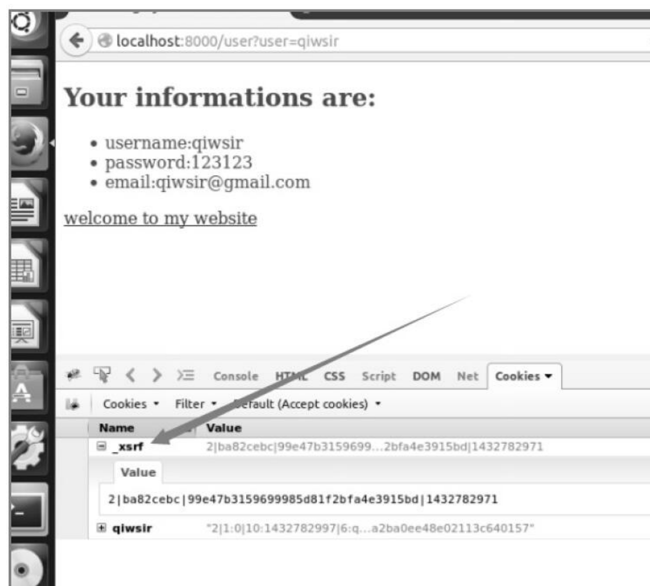


图8-19　运行结果

这是Tornado提供的XSRF防护方法。是不是这样做就高枕无忧了呢？世界是复杂的，要做好一个网站，需要考虑的事情还很多。

常常听到人说做个网站怎么简单，客户用这种说辞来压低价格，老板用这种说辞来缩短工时成本，从上面的简单叙述中，你还觉得网站是随便几个页面就完事儿的吗？除非那个网站不是给人看的，而是在那里摆着的。

8.3.13　用户验证

用户登录之后，当翻到别的网页中时，往往需要验证用户是否处于登录状态。当然，一种比较直接的方法，就是在转到每个目录时，都从Cookie中把用户信息传到后端，跟数据库验证。这不仅是直接的，也是基本的流程。但是，如果这个过程总让用户自己来做，框架的作用就显不出来了。Tornado就提供了一种用户验证方法。

为了后面更工程化地使用Tornado编程，需要将前面已经有的代码进行重新梳理。下面只是将有修改的文件代码写出来，不做过多解释，必要的有注释，相信读者在学习前面内容的基础上能够理解。

在handler目录中增加一个文件，名称是base.py，代码如下：

```
#! /usr/bin/env python
# coding=utf-8

import tornado.web

class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("user")
```

在这个文件中，目前只做一件事情，就是建立一个名为BaseHandler的类，然后在里面放置一个方法，就是得到当前的Cookie。在这里要特别向读者说明，在这个类中，其实还可以写很多别的东西，比如你可以将数据库连接写到这个类的初始化_init_()方法中。因为在其他的类中，我们要继承这个类。所以，这样一个架势就为读者以后的扩展增加了冗余空间。

然后把index.py文件改写为：

```
#!/usr/bin/env python
# coding=utf-8

import tornado.escape
import methods.readdb as mrd
from base import BaseHandler

class IndexHandler(BaseHandler):                          #继承base.py中的
    def get(self):
        usernames = mrd.select_columns(table="users",column="usern
        one_user = usernames[0][0]
        self.render("index.html", user=one_user)

    def post(self):
        username = self.get_argument("username")
        password = self.get_argument("password")
        user_infos = mrd.select_table(table="users", column="*", c
        if user_infos:
            db_pwd = user_infos[0][2]
            if db_pwd == password:
                self.set_current_user(username)    #将当前用户名写入
                self.write(username)
            else:
                self.write("-1")
        else:
            self.write("-1")
```

```
    def set_current_user(self, user):
        if user:
            #注意这里使用了tornado.escape.json_encode()方法
            self.set_secure_cookie('user', tornado.escape.json_enc
        else:
            self.clear_cookie("user")

class ErrorHandler(BaseHandler):                              #增加了一个专门用来
    def get(self):
        self.render("error.html")
```

在index.py的类IndexHandler中，继承了BaseHandler类，并且增加了一个方法，set_current_user()用于将用户名写入Cookie。请读者特别注意tornado.escape.json_encode()方法，其功能是：

```
tornado.escape.json_encode(value) JSON-encodes the given Python ob
```

如果要查看源码，可以阅读：http://www.tornadoweb.org/en/branch2.3/escape.html。

这样做的本质是把user转化为json，写入到了Cookie中。如果从Cookie中把它读出来，使用user的值时，还会用到：

```
tornado.escape.json_decode(value) Returns Python objects for the g
```

它们与json模块中的dump()、load()功能相仿。

接下来要对user.py文件也进行重写：

```
#!/usr/bin/env python
# coding=utf-8

import tornado.web
import tornado.escape
import methods.readdb as mrd
from base import BaseHandler

class UserHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
        #username = self.get_argument("user")
        username = tornado.escape.json_decode(self.current_user)
        user_infos = mrd.select_table(table="users", column="*", c
        self.render("user.html", users = user_infos)
```

在get()方法前面添加@tornado.web.authenticated，这是一个装饰器，它的作用就是完成Tornado的认证功能，即能够得到当前合法用户。在原来的代码中，用username＝self.get_argument（"user"）方法，从url中得到当前用户名，现在把它注释掉，改用self.current_user，这是和前面的装饰器配合使用的，如果它的值为假，就根据setting中的设置，寻找login_url所指定的目录（请关注下面对setting的配置）。

由于在index.py文件的set_current_user()方法中，是将user值转化为json写入Cookie的，这里就得用username＝tornado.escape.json_decode（self.current_user）解码。得到的username值，可以被用于后一句中的数据库查询。

application.py中的setting也要做相应修改：

```
#!/usr/bin/env python
# coding=utf-8

from url import url

import tornado.web
import os

setting = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templ
    static_path = os.path.join(os.path.dirname(__file__), "statics
    cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E="
    xsrf_cookies = True,
    login_url = '/',
)

application = tornado.web.Application(
    handlers = url,
    **setting
)
```

与以前代码的重要区别在于“login_url＝'/'，”，如果用户不合法，根据这个设置，会返回到首页。当然，如果有单独的登录界面，比如/login，也可以login_url＝'/login'。

如此完成的是用户登录到网站之后，在页面转换的时候实现用户认证。

8.3.14 相关概念

1.同步和异步

有不少资料对这两个概念做了不同角度和层面的解释。在我来看，最典型的例子就是打电话和发短信。

打电话就是同步。张三给李四打电话，张三说：“是李四吗？”。当这个信息被张三发出，提交给李四，等待李四的响应（一般会听到“是”，或者“不是”），只有得到了李四返回的信息之后，才能进行后续的信息传送。

发短信是异步。张三给李四发短信，编辑了一句话“今晚一起看老齐的《零基础学python》”，发送给李四。李四或许马上回复，或许过一段时间才回复，这段时间有多长不一定。总之，李四不管什么时候回复，张三会以听到短信铃声为提示查看短信。

以上方式理解“同步”和“异步”不是很精准，有些地方或许有牵强。要严格理解，需要用严格一点的定义表述（以下表述参照了“知乎”上的回答）：

同步和异步关注的是消息通信机制（synchronous communication/asynchronous communication）。

所谓同步，就是在发出一个“调用”时，在没有得到结果之前，该“调用”就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由“调用者”主动等待这个“调用”的结果。

而异步则相反，“调用”在发出之后，就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在“调用”发出后，“被调用者”通过状态、通知来通知调用者，或通过回调函数处理这个调用。

可能还是前面的打电话和发短信更好理解。

2.阻塞和非阻塞

“阻塞和非阻塞”与“同步和异步”常常被混为一谈，其实它们之间还是有差别的。如果按照一个“差不多”先生的思维方法，你也可以不那么深究它们之间学理上的差距，反正在你的程序中，会使用就可以

了。不过，必要的严谨还是需要的，特别是本书中，要装扮的让别人看来自己懂，于是就再引用知乎上的说明：

阻塞和非阻塞关注的是程序在等待调用结果（消息、返回值）时的状态。

阻塞调用是指调用结果返回之前，当前线程会被挂起，调用线程只有在得到结果之后才会返回。非阻塞调用是指在不能立刻得到结果之前，该调用不会阻塞当前线程。

按照这个说明，发短信显然就是非阻塞，发出去一条短信之后，你利用手机还可以干别的，乃至于再发一条“老齐的课程没意思，还是看PHP刺激”也是可以的。

关于这两组基本概念的辨析，不是本教程的重点，读者可以参阅这篇文章：http://www.cppblog.com/converse/archive/2009/05/13/82879.html，文章作者做了细致入微的辨析。

8.3.15　Tornado的同步

此前，在Tornado基础上已经完成的Web就是同步的、阻塞的。为了更明显地感受这一点，不妨这样试一试。

在handlers文件夹中建立一个文件，命名为sleep.py。

```
#!/usr/bin/env python
# coding=utf-8

from base import BaseHandler

import time

class SleepHandler(BaseHandler):
    def get(self):
        time.sleep(17)
        self.render("sleep.html")

class SeeHandler(BaseHandler):
    def get(self):
        self.render("see.html")
```

sleep.html和see.html是两个简单的模板，内容可以自己写。别忘

记修改url.py中的目录。

然后测试稍微复杂一点，打开浏览器之后，打开两个标签，分别在两个标签中输入localhost:8000/sleep（记为标签1）和localhost:8000/see（记为标签2），注意我用的是8000端口。输入之后先不要单击回车访问。做好准备，记住切换标签可以用“ctrl-tab”组合键。

1.执行标签1，让它访问网站。

2.马上切换到标签2，访问网址。

3.注意观察，两个标签页面，是不是都在显示“正在访问，请等待”。

4.当标签1不呈现等待提示（比如一个正在转的圆圈）时，标签2的表现如何？几乎同时也访问成功了。

建议读者修改sleep.py中的time.sleep（17）这个值，多试试。

当然，这是比较笨拙的方法，可以通过测试工具完成上述操作比较。

8.3.16　异步设置

Tornado本来就是一个异步的服务框架，体现在Tornado的服务器和客户端的网络交互的异步上，起作用的是tornado.ioloop.IOLoop。但是如果在客户端请求服务器之后，在执行某个方法的时候，比如上面的代码中执行get()方法的时候，遇到了time.sleep（17）这个需要执行时间比较长的操作，耗费时间，就会使整个Tornado服务器的性能受限。

为了解决这个问题，Tornado提供了一套异步机制，就是异步装饰器@tornado.web.asynchronous。

```
#!/usr/bin/env python
# coding=utf-8

import tornado.web
from base import BaseHandler
```

```
import time

class SleepHandler(BaseHandler):
    @tornado.web.asynchronous
    def get(self):
        tornado.ioloop.IOLoop.instance().add_timeout(time.time() +
    def on_response(self):
        self.render("sleep.html")
        self.finish()
```

将sleep.py的代码如上述一样改造，即在get()方法前面增加了装饰器@tornado.web.asynchronous，它的作用在于将Tornado服务器本身默认的设置_auto_fininsh值修改为False。如果不用这个装饰器，客户端访问服务器的get()方法并得到返回值之后，两者之间的连接就断开了，但是用了@tornado.web.asynchronous之后，这个连接就不关闭，直到执行了self.finish()才关闭这个连接。

tornado.ioloop.IOLoop.instance().add_timeout()也是一个实现异步的函数，time.time()＋17给前面的函数提供一个参数，这样实现了相当于time.sleep（17）的功能，不过，还没有完成，当这个操作完成之后，就执行回调函数on_response()中的self.render（"sleep.html"），并关闭连接self.finish()。

所谓异步，就是要解决原来的time.sleep（17）造成的服务器处理时间长、性能下降的问题。解决方法如上描述。

读者看这个代码，或许会感觉有点不舒服。如果有这个感觉是正常的，因为它里面除了装饰器之外，用到了一个回调函数，它让代码的逻辑不是平铺下去，而是被分割成了两段。第一段是tornado.ioloop.IOLoop.instance().add_timeout（time.time()＋17，callback＝self.on_response），用callback＝self.on_response来使用回调函数，并没有如同改造之前直接self.render（"sleep.html"）；第二段是回调函数on_response（self），要在这个函数里面执行self.render（"sleep.html"），并且以self.finish()`结尾以关闭连接。

这还是执行简单逻辑，如果复杂了，要不断地进行“回调”，无法让逻辑顺利延续，就会“眩晕”了。这种现象被业界称为“代码逻辑拆分”，打破了原有逻辑的顺序性。为了让代码逻辑不至于被拆分的七零八落，于是就出现了另外一种常用的方法：

```
#!/usr/bin/env python
# coding=utf-8
```

```
import tornado.web
import tornado.gen
from base import BaseHandler

import time

class SleepHandler(tornado.web.RequestHandler):
    @tornado.gen.coroutine
    def get(self):
        yield tornado.gen.Task(tornado.ioloop.IOLoop.instance().ad
        #yield tornado.gen.sleep(17)
        self.render("sleep.html")
```

从整体上看，这段代码避免了回调函数，看着顺利多了。

再看细节部分。

首先使用的是@tornado.gen.coroutine装饰器，所以要在前面有import tornado.gen。跟这个装饰器类似的是@tornado.gen.engine装饰器，两者功能类似，有一点细微差别。请阅读官方对此的解释：

This decorator（指engine）is similar to coroutine，except it does not return a Future and the callback argument is not treated specially.

@tornado.gen.engine是古时候用的，现在我们都使用@tornado.gen.corroutine了，这个是在tornado 3.0以后开始用的。在网上查阅资料的时候，会遇到一些使用@tornado.gen.engine的，但是在你使用或者借鉴代码的时候，可以勇敢地将其修改为@tornado.gen.coroutine。有了这个装饰器，就能够控制下面的生成器的流程了。

然后就看到get()方法里面的yield了，这是一个生成器。“yield tornado.gen.Task（tornado.ioloop.IOLoop.instance().add_timeout，time.time()＋17）”的执行过程，先看括号里面，跟前面的一样是来替代time.sleep（17）的，然后是tornado.gen.Task()方法，其作用是“Adapts a callback-based asynchronous function for use in coroutines.”（由于怕翻译后遗漏信息，所以引用原文）。返回后，使用yield得到了一个生成器，先把流程挂起，等完全完毕再唤醒继续执行。要提醒读者，生成器都是异步的。

其实，上面啰嗦了一堆，可以用代码中注释的一句话来代替yield tornado.gen.sleep（17），之所以啰嗦，就是为了顺便看到

tornado.gen.Task()方法，因为如果读者在看古老的代码时会遇到。但是，后面你写的时候，就不要那么啰嗦了，请用yield tornado.gen.sleep()。

至此，基本上对Tornado的异步设置有了概览，不过，上面的程序在实际中没有什么价值。在工程中，要让Tornado网站真正异步起来还要做很多事情，不仅仅是如上面的设置，因为其实很多东西都不是异步的。

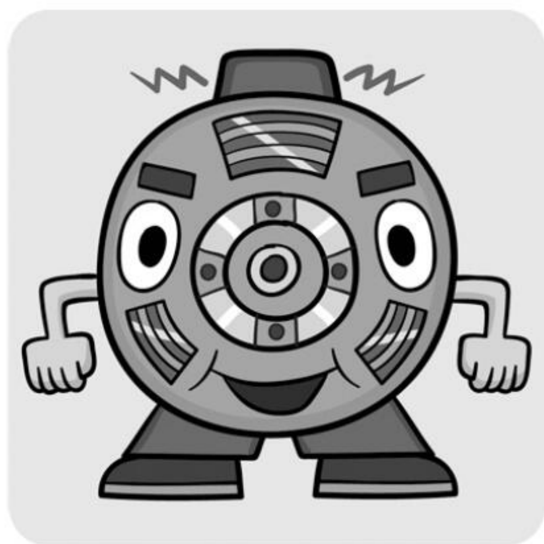
在研发实践中，异步设置是比较复杂的，不是简单地完成上述流程就行了。比如以下各项中，尽管你已经完成了前面的设置，如果忽视了下面这些项目，那么tornado的非阻塞、异步优势削减了。

- 数据库的所有操作，不管你的数据是SQL还是noSQL、connect、insert、update等。
- 文件操作，打开、读取、写入等。
- time.sleep，在前面举例中已经看到了。
- smtplib，发邮件的操作。
- 一些网络操作，比如tornado的httpclient以及pycurl等。

或许在编程实践中还会遇到其他的同步、阻塞实践。仅仅就上面几项，是编程实践中经常会遇到的，怎么解决？

聪明的大牛程序员帮我们做了扩展模块，专门用来实现异步/非阻塞。

- 在数据库方面，由于种类繁多，不能一一说明，比如MySQL，可以使用adb模块来实现Python的异步MySQL库；对于mongodb数据库，有一个非常优秀的模块，专门用于在Tornado和mongodb上实现异步操作，它就是motor。下面特别贴出它的Logo。
- 文件操作方面也没有替代模块，只能尽量控制好IO，或者使用内存型（Redis）及文档型（MongoDB）数据库。
- time.sleep()在Tornado中有替代：tornado.gen.sleep()或者tornado.ioloop.IOLoop.instance().add_timeout，这在前面代码已经显示了。
- smtp发送邮件，推荐改为tornado-smtp-client。
- 对于网络操作，要使用tornado.httpclient.AsyncHTTPClient。



其他的解决方法，只能看到问题具体说了，甚至没有很好的解决方法。不过，这里有一个列表，列出了足够多的库，供使用者选择：Async Client Libraries built on tornado.ioloop（https://github.com/tornadoweb/tornado/wiki/Links），同时这个页面里面还有很多别的链接，都是很好的资源，建议读者多看看。

到这里，请读者思考一个问题，既然对于mongodb有专门的motor库来实现异步，前面对于Tornado的异步，不管是哪个装饰器，都感觉麻烦，有没有专门的库来实现这种异步呢？这不是异想天开，还真有。也应该有，因为这才体现Python的特点。比如greenlet-tornado（https://github.com/mopub/greenlet-tornado）就是一个不错的库。读者可以浏览官方网站深入了解。

必须声明，前面演示如何在Tornado中设置异步的代码，仅仅是演示设置方法。在工程实践中，那个代码的意义不大，为此，应该有一个近似于实践的代码示例。是的，的确应该有。当我正要写这样的代码时，在网上发现一篇文章，这篇文章阻止了我写下去，因为我要写的内容那篇文章的作者早就写好了，而且我认为表述非常到位，示例也详细。所以，我不得不放弃，转而推荐给读者这篇好文章：http://emptysqua.re/blog/refactoring-tornado-coroutines/。

第9章　科学计算

有朋友问，Python在哪个方面能让我感到最强悍？我回答：计算。如果说仅仅要做一个实现增删改查功能的网站，PHP也是很好的选择，Python也没有什么太让人惊叹的表现。但若要进行计算，特别是在数据分析、机器学习等方面，Python的表现会让人佩服得五体投地。

因为本书的读者是初学者，这里所谓的“科学计算”，仅仅是给读者展示一个简单的样子，要专业研究用Python进行科学计算，还请读者参阅有关专门的书籍。

9.1　为计算做准备

9.1.1　闲谈

计算机姑娘是擅长进行科学计算的，本来她就是做这个的，只不过后来人们让她处理了很多文字内容罢了，乃至于现在有一些人认为她是用来打字写文章的，却忘记了她最擅长的计算。

每种编程语言都能用来做计算，区别是在编程过程中是否有足够的工具包供给。比如，用汇编就得自己多劳动，如果用Fortran，就方便多了。不知道读者是否听说过Fortran，貌似古老，但现在仍被使用（以下引文均来自维基百科）。

Fortran语言是为了满足数值计算的需求而发展出来的。1953年12月，IBM公司工程师约翰·巴科斯（J.Backus）因深深体会编写程序很困难，而写了一份备忘录给董事长斯伯特·赫德（Cuthbert Hurd），建议为IBM704系统设计全新的计算机语言以提升开发效率。当时IBM公司的顾问冯·诺伊曼强烈反对，因为他认为不切实际而且根本不必要。但赫德批准了这项计划。1957年，IBM公司开发出第一套Fortran语言，在IBM704计算机上运作。历史上第一支Fortran程序在马里兰州的西屋贝地斯核电厂试验。1957年4月20日星期五的下午，一位IBM软件工程师决定在电厂内编译第一支Fortran程序，当程序代码输入后，经过编译，打印机列出一行讯息："原始程序错误……右侧括号后面没有逗号"，这让现场人员都感到惊讶，修正这个错误后，打印机输出了正确结果。而西屋电气公司因此意外地成为Fortran的第一个商业用户。1958年推出FortranII，几年后又推出FortranIII，在1962年推出FortranIV后，开始被广泛使用。目前最新版是Fortran 2008。

还有一个被广为应用的语言不得不说，那就是MATLAB，一直以来被人称赞。

MATLAB（矩阵实验室）是MATrix LABoratory的缩写，是一款由美国The MathWorks公司出品的商业数学软件。MATLAB是一种用于算法开发、数据可视化、数据分析以及数值计算的高级技术计算语言和交互式环境。除了矩阵运算、绘制函数/数据图像等常用功能

外，MATLAB还可以用来创建用户界面及调用其他语言（包括C、C++、Java、Python和Fortran）编写的程序。

但是，它是收费的商业软件，虽然在个别国家并不受到收费影响。

还有R语言，也是在计算领域被广泛使用的。

R语言，一种自由软件程序语言与操作环境，主要用于统计分析、绘图、数据挖掘。R本来是由来自新西兰奥克兰大学的Ross Ihaka和Robert Gentleman开发（也因此称为R），现在由“R开发核心团队”负责开发。R是基于S语言的一个GNU计划项目，所以也可以当作S语言的一种实现，通常用S语言编写的代码都可以不做修改地在R环境下运行。R的语法是来自Scheme。

最后要说的就是Python，近几年使用Python的领域不断扩张，包括在科学计算领域，它已经成为了一种趋势。在这个过程中，虽然有不少人诟病Python慢、解释动态语言之类（这种说法是值得讨论的），但依然无法阻挡Python在科学计算领域大行其道。之所以这样，就是因为它是Python，天生骄傲。

- 开源，就这一条已经足够了，一定要用开源的东西。
- 因为开源，所以有非常棒的社区，里面有相当多支持科学计算的库。
- 简单易学，这一点对那些不是专业的程序员来讲非常重要。接触过一些搞天文学和生物学的研究者，他们也正在使用Python进行计算。
- 在科学计算上如果使用Python，能够让数据与其他领域比如Web无缝对接。

当然，最重要一点是本书是讲Python的，所以，在科学计算这块肯定不会讲Fortran或者R，而是会讲Python。

9.1.2　安装

如果读者使用Ubuntu或者Debian，可以这样来安装：

sudo apt-get install python-numpy python-scipy python-matplotlib ipython ipython-notebook python-pandas python-sympy python-nose

一股脑把可能用上的都先装上。在安装的时候，如果需要其他依赖，你会明显看到的。

如果是别的系统，比如Windows，请自己去网上查找安装方法吧，最权威的是看官方网站列出的安装：http://www.scipy.org/install.html。

9.1.3　基本操作

在科学计算中，业界比较喜欢使用ipython notebook，前面已经有安装。在shell中执行：

ipython notebook--pylab＝inline

得到如图9-1所示的界面，这是在浏览器中打开的。



图9-1　在浏览器中打开界面

在In后面的编辑区，可以写Python语句。然后按下SHIFT＋ENTER或者CTRL＋ENTER组合键就能执行了，如果按下ENTER键，则不是执行，而是在当前编辑区换行。

Ipython Notebook是一个非常不错的编辑器，执行之后，直接显示出来输入的内容和输出的结果。当然，错误是难免的，如图9-2所示。

```
In [9]: import random
        a = [random.randint(10)]
        print a

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        /home/qw/Documents/StarterLearningPython/<ipython-input-9-44c3f4780939> in <module>()
              1 import random
        ----> 2 a = [random.randint(10)]
              3 print a

        TypeError: randint() takes exactly 3 arguments (2 given)
```

图9-2　显示内容

注意观察图中的箭头所指，直接标出有问题的行。返回编辑区，修改之后可继续执行。

不要忽视左边的辅助操作，它能够让你在使用ipython notebook的时候更方便，如图9-3所示。

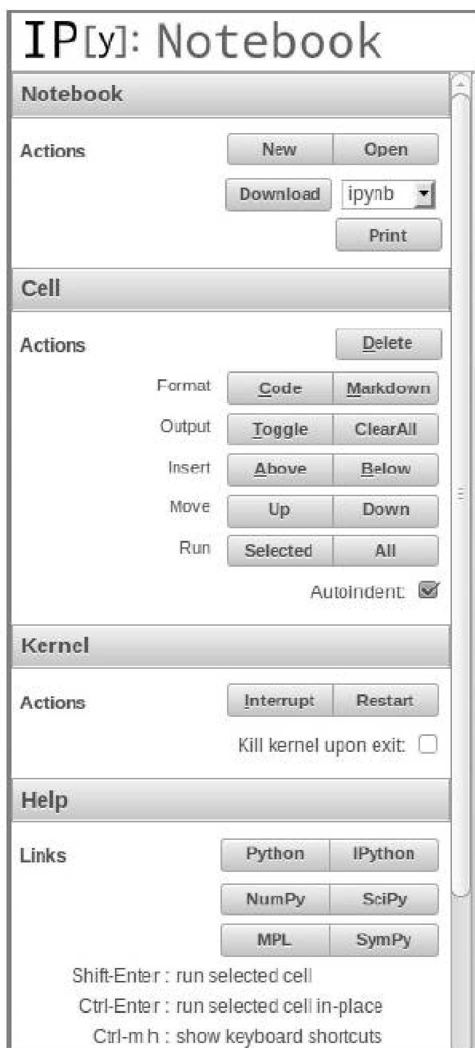


图9-3　左边的辅助操作

除了在网页中之外，如果你已经喜欢上了Python的交互模式，特别是你用的计算机中有shell，那就更棒了。于是可以：

```
$ ipython
```

进入了一个类似于Python的交互模式中，如下所示：

```
In [1]: print "hello, pandas"
hello, pandas

In [2]:
```

或者说ipython同样是一个不错的交互模式。

9.2 Pandas

Pandas是基于NumPy的一个非常好用的库，正如名字一样，人见人爱。之所以如此，是因为不论是读取还是处理数据，用它都非常简单。

9.2.1 基本的数据结构

Pandas有两种自己独有的基本数据结构。读者应该注意的是，它固然有着两种数据结构，因为它依然是Python的一个库，所以，Python中有的数据类型在这里依然适用，同样还可以使用类自己定义数据类型。不过，Pandas里面又定义了两种数据类型：Series和DataFrame，它们让数据操作更简单。

以下操作都是基于如下模块导入：

```
In [1]: from pandas import Series, DataFrame
        import pandas as pd
```

为了省事，后面就不再显示了。如果你跟我一样是使用的ipython notebook，只需要开始引入模块即可。

- Series

Series如同列表一样，有一系列数据，每个数据对应一个索引值。比如这样一个列表：[9，3，8]，如果跟索引值写到一起，就是：

data	9	3	8
index	0	1	2

这种样式我们已经熟悉了，不过，有些时候需要把它竖过来表示：

index	data
0	9
1	3
2	8

上面两种，只是表现形式上的差别罢了。

Series就是“竖起来”的list：

```
In [2]: s = Series([100, "PYTHON", "Soochow", "Qiwsir"])
        s

Out[2]: 0        100
        1     PYTHON
        2    Soochow
        3     Qiwsir
```

另外一点也很像列表，就是其中元素的类型由你任意决定（其实是由需要来决定）。

这样，我们就创建了一个Series对象，这个对象有其属性和方法。比如，下面的两个属性依次可以显示Series对象的数据值和索引：

```
In [3]: s.values

Out[3]: array([100, 'PYTHON', 'Soochow', 'Qiwsir'], dtype=object)

In [4]: s.index

Out[4]: Int64Index([0, 1, 2, 3], dtype=int64)
```

列表的索引只能是从0开始的整数，在默认情况下，Series数据类型其索引也是如此。但区别于列表的是，Series可以自定义索引：

```
s2 = Series([100, "PYTHON", "Soochow", "Qiwsir"], index=["mark", "title", "university", "name"])
s2

mark              100
title          PYTHON
university    Soochow
name           Qiwsir
```

每个元素都有了索引，就可以根据索引操作元素了。还记得list中的操作吗？在Series中也有类似的操作。先看简单的，根据索引查看其值和修改其值：

```
In [7]: s2["name"]

Out[7]: 'Qiwsir'
```

```
In [8]: s2["name"] = "AOI"
```

```
In [9]: s2

Out[9]: mark                100
        title            PYTHON
        university      Soochow
        name                AOI
```

这是不是又有点类似dict数据？的确如此，看下面就理解了。

前面定义Series对象的时候，用的是列表，即Series()方法的参数中第一个列表就是其数据值，如果需要定义index，放在后面依然是一个列表。除了这种方法之外，还可以用下面的方法定义Series对象：

```
In [15]: sd = {"python":8000, "c++":8100, "c#":4000}
         s4 = Series(sd)
         s4

Out[15]: c#        4000
         c++       8100
         python    8000
```

现在是否理解为什么前面那个类似dict了？因为本来就是可以这样定义的。

这时候，索引依然可以自定义。Pandas的优势在这里体现出来，如果自定义了索引，自定义的索引会自动寻找原来的索引。如果一样，就取原来索引对应的值，这个可以简称为“自动对齐”。

```
In [19]: s6 = Series(sd, index=["java","python","c++","c#"])
         s6

Out[19]: java       NaN
         python    8000
         c++       8100
         c#        4000
```

在sd中，只有“'python':8000，'c＋＋':8100，'c#':4000”，没有“java”，但是在索引参数中有，于是其他能够“自动对齐”的照搬原值，没有的那个“java”依然在新Series对象的索引中存在，并且自动为

其赋值NaN。在Pandas中，如果没有值，都对齐赋给NaN。下面来一个更特殊的：

```
In [17]: ilst = ["java", "perl"]
         s5 = Series(sd,index=ilst)
         s5

Out[17]: java    NaN
         perl    NaN
```

新得到的Series对象索引与sd对象一个也不对应，所以都是NaN。

Pandas有专门的方法来判断值是否为空。

```
In [20]: pd.isnull(s6)

Out[20]: java       True
         python    False
         c++       False
         c#        False

In [21]: pd.notnull(s6)

Out[21]: java      False
         python     True
         c++        True
         c#         True
```

此外，Series对象也有同样的方法：

```
In [22]: s6.isnull()

Out[22]: java       True
         python    False
         c++       False
         c#        False
```

其实，对索引的名字是可以重新定义的：

```
In [38]: s6.index = ["p1", "p2", "p3", "p4"]
         s6

Out[38]: p1     NaN
         p2    8000
         p3    8100
         p4    4000
```

对于Series数据，也可以做类似下面的运算：

```
In [10]: s3 = Series([3,9,4,7], index=['a','b','c','d'])
         s3

Out[10]: a    3
         b    9
         c    4
         d    7

In [11]: s3[s3 > 5]

Out[11]: b    9
         d    7

In [12]: s3 * 5

Out[12]: a    15
         b    45
         c    20
         d    35
```

上面的演示都是在ipython notebook中进行的，所以截图了。在学习Series数据类型的同时了解了ipyton notebook。对于后面的所有操作，读者都可以在ipython notebook中进行，也可以在Python交互模式中进行。

- DataFrame

DataFrame是一种二维的数据结构，非常接近于电子表格或者类似MySQL数据库的形式。它的竖行称之为columns，横行跟前面的Series一样，称之为index，也就是说可以通过columns和index来确定一个主句的位置。

	c1	c2	c3	c4	columns
ix1					
ix2					
ix3		(c2, ix3)			
ix4					
index					

下面的演示在Python交互模式下进行，读者仍然可以在ipython notebook环境中测试。

```
>>> import pandas as pd
>>> from pandas import Series, DataFrame

>>> data = {"name":["yahoo","google","facebook"], "marks":[200,400
>>> f1 = DataFrame(data)
>>> f1
     marks  name      price
0    200    yahoo     9
1    400    google    3
2    800    facebook  7
```

这是定义一个DataFrame对象的常用方法——使用dict定义。字典的“键”（"name"，"marks"，"price"）就是DataFrame的columns的值（名称），字典中每个“键”的“值”是一个列表，它们就是那一竖列中的填充数据。上面的定义中没有确定索引，所以，按照惯例（Series中已经形成的惯例）就是从0开始的整数。从上面的结果中很明显表示出来，这就是一个二维的数据结构（类似Excel或者MySQL中的查看效果）。

上面的数据显示中，columns的顺序没有规定，就如同字典中键的顺序一样，但是在DataFrame中，columns跟字典键相比，有一个明显不同，就是其顺序可以被规定，如下所示：

```
>>> f2 = DataFrame(data, columns=['name','price','marks'])
>>> f2
       name     price  marks
0     yahoo     9      200
1    google     3      400
2  facebook     7      800
```

跟Series类似，DataFrame数据的索引也能够自定义。

```
>>> f3 = DataFrame(data, columns=['name', 'price', 'marks', 'debt'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/pymodules/python2.7/pandas/core/frame.py", line 2
    mgr = self._init_dict(data, index, columns, dtype=dtype)
  File "/usr/lib/pymodules/python2.7/pandas/core/frame.py", line 3
    mgr = BlockManager(blocks, axes)
  File "/usr/lib/pymodules/python2.7/pandas/core/internals.py", li
    self._verify_integrity()
  File "/usr/lib/pymodules/python2.7/pandas/core/internals.py", li
    assert(block.values.shape[1:] == mgr_shape[1:])
AssertionError
```

报错了。这个报错信息太不友好了，也没有提供什么线索，这就是交互模式的不利之处。修改之，错误在于index的值——列表，其数据项多了一个，data中是三行，这里给出了四个项（['a'，'b'，'c'，'d']）。

```
>>> f3 = DataFrame(data, columns=['name', 'price', 'marks', 'debt'
>>> f3
       name      price  marks  debt
a     yahoo      9      200     NaN
b    google      3      400     NaN
c  facebook      7      800     NaN
```

读者还要注意观察上面的显示结果。因为在定义f3的时候，columns的参数中比以往多了一项（'debt'），但是这项在data这个字典中并没有，所以debt这一竖列的值都是空的，在Pandas中，空就用NaN来代表了。

定义DataFrame的方法，还可以使用“字典套字典”的方式。

```
>>> newdata = {"lang":{"firstline":"python","secondline":"java"},
>>> f4 = DataFrame(newdata)
>>> f4
              lang     price
firstline     python   8000
secondline    java     NaN
```

在字典中就规定好数列名称（第一层键）和每横行索引（第二层字典键）以及对应的数据（第二层字典值），即在字典中规定好每个数据格子中的数据，没有规定的都是空。

```
>>> DataFrame(newdata, index=["firstline","secondline","thirdline"
              lang     price
firstline     python   8000
secondline    java     NaN
thirdline     NaN      NaN
```

如果额外确定了索引，就如同上面显示的一样，除非在字典中有相应的索引内容，否则都是NaN。

前面定义了DataFrame数据，它也是一种对象类型，比如变量f3引用了一个对象，它的类型是DataFrame。承接以前的思维方法：对象有属性和方法。

```
>>> f3.columns
Index(['name', 'price', 'marks', 'debt'], dtype=object)
```

DataFrame对象的columns属性能够显示素有的columns名称，并且，还能用下面类似字典的方式得到某竖列的全部内容（当然包含索引）：

```
>>> f3['name']
a         yahoo
b        google
c      facebook
Name: name
```

这其实就是一个Series，或者说，可以将DataFrame理解为是由一个一个的Series组成的。

一直耿耿于怀没有数值的那一列，下面的操作是统一给那一列赋值：

```
>>> f3['debt'] = 89.2
>>> f3
       name     price  marks  debt
a     yahoo     9        200  89.2
b    google     3        400  89.2
c  facebook     7        800  89.2
```

除了能够统一赋值之外，还能够“点对点”添加数值，结合前面的Series，既然DataFrame对象的每竖列都是一个Series对象，那么可以先定义一个Series对象，然后把它放到DataFrame对象中。如下：

```
>>> sdebt = Series([2.2, 3.3], index=["a","c"])     #注意索引
>>> f3['debt'] = sdebt
```

将Series对象（sdebt变量所引用）赋给f3['debt']列，Pandas的一个重要特性——自动对齐，在这里起做用了，在Series中，只有两个索引（"a"，"c"），它们将和DataFrame中的索引自动对齐。于是乎：

```
>>> f3
       name  price  marks  debt
a     yahoo  9        200   2.2
b    google  3        400   NaN
c  facebook  7        800   3.3
```

自动对齐之后，没有被复制的依然保持NaN。

还可以更精准地修改数据吗？当然可以，完全仿照字典的操作：

```
>>> f3["price"]["c"]= 300
>>> f3
       name   price   marks  debt
a     yahoo   9       200    2.2
b    google   3       400    NaN
c  facebook   300     800    3.3
```

这些操作是不是都不陌生？这就是Pandas中的两种数据对象。

9.2.2　读取CSV文件

因为篇幅限制，不能将有关Pandas的内容完全详细讲述，只能“抛砖引玉”，向大家做一个简单介绍，说明其基本使用方法。当读者在实践中使用的时候，如果遇到问题，可以结合相关文档或者通过搜索来解决。

1.关于CSV文件

CSV是一种通用的、相对简单的文件格式，在表格类型的数据中用途很广泛，很多关系型数据库都支持这种类型文件的导入导出，并且Excel这种常用的数据表格也能和CSV文件之间转换。

逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。纯文本意味着该文件是一个字符序列，不含必须像二进制数字那样被解读的数据。CSV文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间的分隔符是其他字符或字符串，最常见的是逗号或制表符。通常，所有记录都有完全相同的字段序列。

从上述维基百科的叙述中，重点要解读出“字段间分隔符”和“最常见的是逗号或制表符”，当然，这种分隔符也可以自行制定。比如下面这个我命名为marks.csv的文件，就是用逗号（必须是半角的）作为分隔符：

```
name,physics,python,math,english
Google,100,100,25,12
Facebook,45,54,44,88
Twitter,54,76,13,91
Yahoo,54,452,26,100
```

其实，这个文件要表达的事情是（如果转化为表格形式）：

	A	B	C	D	E
1	name	physics	python	math	english
2	Google	100	100	25	12
3	Facebook	45	54	44	88
4	Twitter	54	76	13	91
5	Yahoo	54	452	26	100
6					

最简单、最直接的就是用open()打开文件：

```
>>> with open("./marks.csv") as f:
...     for line in f:
...         print line
...
name,physics,python,math,english

Google,100,100,25,12

Facebook,45,54,44,88

Twitter,54,76,13,91

Yahoo,54,452,26,100
```

此方法可以，但略显麻烦。

Python中还有一个CSV的标准库，足可见CSV文件的使用频繁了。

```
>>> import csv
>>> dir(csv)
['Dialect', 'DictReader', 'DictWriter', 'Error', 'QUOTE_ALL', 'QUO
```

什么时候也不要忘记这种最佳学习方法。从上面的结果可以看出CSV模块提供的属性和方法。仅仅就读取本例子中的文件：

```
>>> import csv
```

```
>>> csv_reader = csv.reader(open("./marks.csv"))
>>> for row in csv_reader:
...     print row
...
['name', 'physics', 'python', 'math', 'english']
['Google', '100', '100', '25', '12']
['Facebook', '45', '54', '44', '88']
['Twitter', '54', '76', '13', '91']
['Yahoo', '54', '452', '26', '100']
```

算是稍有改善。

如果对上面的结果都不满意的话，那么看看Pandas的效果：

```
>>> import pandas as pd
>>> marks = pd.read_csv("./marks.csv")
>>> marks
       name  physics  python  math  english
0    Google      100     100    25       12
1  Facebook       45      54    44       88
2   Twitter       54      76    13       91
3     Yahoo       54     452    26      100
```

看了这样的结果，你还不感到惊讶吗？你还不喜欢Pandas吗？这是多么精妙的显示，它就是一个DataFrame数据。

还有另外一种方法：

```
>>> pd.read_table("./marks.csv", sep=",")
       name  physics  python  math  english
0    Google      100     100    25       12
1  Facebook       45      54    44       88
2   Twitter       54      76    13       91
3     Yahoo       54     452    26      100
```

如果你有足够的好奇心来研究这个名叫DataFrame的对象，可以这样：

```
>>> dir(marks)
['T', '_AXIS_ALIASES', '_AXIS_NAMES', '_AXIS_NUMBERS', '__add__',
```

一个一个浏览一下，通过名字可以知道那个方法或者属性的大概，然后就可以根据你自己的喜好和需要，试一试：

```
>>> marks.index
Int64Index([0, 1, 2, 3], dtype=int64)
>>> marks.columns
Index([name, physics, python, math, english], dtype=object)
>>> marks['name'][1]
'Facebook'
```

这几个是让你回忆一下前面的。从DataFrame对象的属性和方法中找一个，再尝试：

```
>>> marks.sort(column="python")
       name  physics  python  math  english
1  Facebook       45      54    44       88
2   Twitter       54      76    13       91
0    Google      100     100    25       12
3     Yahoo       54     452    26      100
```

按照竖列“python”的值排队，结果也是很让人满意的。下面几个操作，也是常用到的，并且秉承了Python的一贯方法：

```
>>> marks[:1]
     name  physics  python  math  english
0  Google      100     100    25       12
>>> marks[1:2]
       name  physics  python  math  english
1  Facebook       45      54    44       88
>>> marks["physics"]
0     100
1      45
2      54
3      54
Name: physics
```

可以说，当你已经掌握了通过dir()和help()查看对象的方法和属性时，就已经掌握了Pandas的用法，其实何止Pandas，其他对象都是如此。

2.读取其他格式数据

CSV是常用来存储数据的格式之一，此外常用的还有MS Excel格式的文件，以及JSON和XML格式的数据等，它们都可以使用Pandas来轻易读取。

在下面的结果中寻觅一下，有没有跟Excel有关的方法。

```
>>> dir(pd)
['DataFrame', 'DataMatrix', 'DateOffset', 'DateRange', 'ExcelFile'
```

虽然没有类似read_csv()的方法，但是有ExcelFile类，于是乎：

```
>>> xls = pd.ExcelFile("./marks.xlsx")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/pymodules/python2.7/pandas/io/parsers.py", line 5
    from openpyxl import load_workbook
ImportError: No module named openpyxl
```

我这里少了一个模块，看报错提示，用pip安装openpyxl模块：sudo pip install openpyxl。继续：

```
>>> xls = pd.ExcelFile("./marks.xlsx")
>>> dir(xls)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
>>> xls.sheet_names
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet1 = xls.parse("Sheet1")
>>> sheet1
   0    1    2   3    4
0  5  100  100  25   12
1  6   45   54  44   88
2  7   54   76  13   91
3  8   54  452  26  100
```

结果中，columns的名字与前面CSV结果不一样，数据部分是同样的结果。从结果中可以看到，sheet1也是一个DataFrame对象。

对于单个的DataFrame对象，如何通过属性和方法进行操作？如果读者理解了本书从一开始就贯穿进来的思想——利用dir()和help()或者到官方网站看文档——此时就能比较轻松地进行各种操作了。

从数据库中查询出来的数据，也可以按照Series或者DataFrame类型数据进行组织，然后就可以对其操作。

9.2.3　处理股票数据

某段时间某国股市很火爆，不少专家在分析股市火爆的各种原

因，不久，该国的股票又开始狂跌，各种各样的救市政策仅仅是螳臂当车罢了。不过，我还是很淡定的，因为没钱。

但是，为了体现本人也是与时俱进的，就以股票数据为例子，来简要说明Pandas和其他模块在处理数据上的应用。

1.下载YAHOO上的数据

或许你好奇，为什么要下载YAHOO上的股票数据呢？国内网站上不是也有吗？有，但我不喜欢用。我喜欢YAHOO，因为它曾经吸引我，注意我说的是www.yahoo.com。



虽然YAHOO的身影渐行渐远，但它终究是值得记忆的。所以，我要演示如何下载YAHOO财经栏目中的股票数据。

```
In [1]: import pandas
In [2]: import pandas.io.data

In [3]: sym = "BABA"

In [4]: finace = pandas.io.data.DataReader(sym, "yahoo", start="20
In [5]: print finace.tail(3)
                 Open       High        Low      Close    Volume
Date
2015-06-17  86.580002  87.800003  86.480003  86.800003  10206100
2015-06-18  86.970001  87.589996  86.320000  86.750000  11652600
2015-06-19  86.510002  86.599998  85.169998  85.739998  10207100
```

下载了阿里巴巴的股票数据（自2014年11月11日以来），并且打印最后三条。

2.画图

已经得到了一个DataFrame对象，就是前面已经下载并用finace变量引用的对象。

```
In[6]: import matplotlib.pyplot as plt
In [7]: plt.plot(finace.index, finace["Open"])
Out[]: [<matplotlib.lines.Line2D at 0xa88e5cc>]

In [8]: plt.show()
```

结果如图9-4所示。

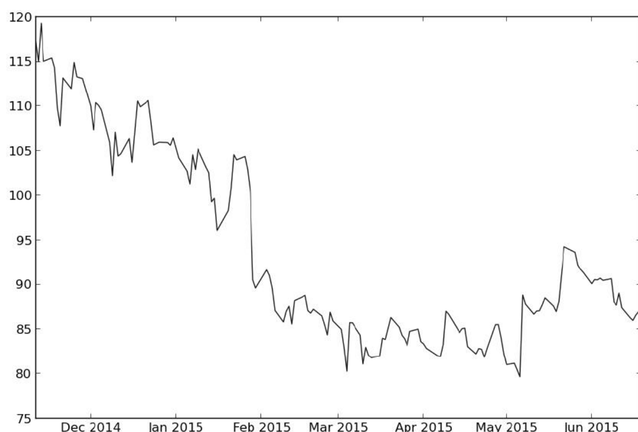


图9-4　阿里巴巴股票数据图

从图9-4中可以看出阿里巴巴的股票自从2014年11月11日到2015年6月19日的股票开盘价的变化。

上面指令中的import matplotlib.pyplot as plt是此前没有看到的。matplotlib模块是Python中绘制二维图形的模块，是最好的模块。可惜matplotlib的发明者——John Hunter已经于2012年8月28日因病医治无效英年早逝，这真是天妒英才呀。为了缅怀他，请读者访问官方网站：matplotlib.org（http://matplotlib.org/），并认真学习这个模块的使用。

经过上面的操作，读者可以用dir()这个以前常用的法宝来查看finace所引用的DataFrame对象的方法和属性等。只要运用dir＋help就能够对这个对象进行操作，也就是能够对该股票数据进行各种操作。

再次声明，本书仅仅是稍微演示一下相关操作，如果读者要深入研习，恭请寻找相关的专业书籍资料阅读学习。